Dense 3D Reconstruction in Dynamic Environments

by

Zhaozhong Chen

B.S., Tianjin University, 2016

M.S., University of Colorado Boulder, 2018

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Electrical, Computer and Energy Engineering

2021

Committee Members:

Prof. Christoffer Heckman, Chair

Prof. Nisar Ahmed

Prof. Marco Nicotra

Prof. Daniel Szafir

Prof. Nikolaus Correll

Chen, Zhaozhong (Ph.D., Electrical Engineering)

Dense 3D Reconstruction in Dynamic Environments

Thesis directed by Prof. Christoffer Heckman

3D reconstruction/dense mapping can be used in many applications such as medical imaging, virtual reality, navigations, etc. In this dissertation, we discussed the different state-of-the-art approaches for 3D reconstruction using visual sensors. Current dense space reconstruction representation can be point cloud, octomap, voxel, surfel, etc. The system generally requires the scan of the environment along with the tracking estimation of the moving sensors. This dissertation provides solutions to the dense 3D reconstruction with visual/visual-inertial sensors in a dynamic environment and tests it both on the CPU and GPU. We decouple the sensor tracking and dense reconstruction. Additionally, we develop a learning-based segmentation technique to detect possible dynamic objects and use the static objects for robust sensor tracking. Then we apply the tracking result to the reconstruction system. We modulize our system so each part (tracking, segmentation, reconstruction) can be replaced in the future by a more advanced tracking/mapping system for better performance. The source code is shared for the benefit of the community[1] .

[1] `https://github.com/arpg/dyna_infini`

Dedication

To all the people who support me, and robots, and sensors.

Acknowledgements

Five years in Boulder, Colorado is a treasure in my lifetime. Thanks my advisor Christoffer Heckman for the help from various aspects. Thanks my parents for all the support across the sea.

Contents

Tables

Table

Figures

Figure

Chapter 1

Introduction

3D reconstruction has attracted a lot of attention due to the evolution of algorithms and hardware in the last decades. An RGB camera can give a 2D image of the environment, and we use 3D reconstruction to rebuild the environment based on the 2D scan. We can see an example from Figure 1.1. It normally requires the camera pose and image depth (not true for some learning-based approaches). We can directly acquire image depth through the RGB-D camera. However, the camera pose is estimated (normally we don't have groundtruth pose) by a computer vision research branch called SLAM (Simultaneous Localization and Mapping) [35], [6]. The offline 3D reconstruction can give a precise result because it took hours, even days, to reconstruct a scene. However, we hope to reconstruct the scene at a milliseconds level in many applications for real-time implementation. Most of the 3D reconstruction projects consider the environment to be static for simplification [117, 76, 118]. However, in applications such as the self-driving car, it is important to consider dynamic objects such as pedestrians and cars. As a result, 3D reconstruction in a dynamic scene has drawn people's attention in recent years. Compared to 3D reconstruction in the static environment, 3D reconstruction in the dynamic environment is more challenging. In recent years, we have some significant work showing in this area, such as Mask Fusion [95], Co-Fusion [96], and DynSLAM [8]. Most of these 3D reconstructions in dynamic environments benefit from deep learning-based image segmentation and object detection. If we want to reconstruct the dynamic scene from a 2D image with people and cars, we naturally hope to know if a pixel corresponds to a person/car, then we can treat that pixel as a "potential" moving object. The game has changed

in this computer vision area since 2015.

Realtime object detection is still not achievable (various science movies give us an illusion that we can do that much earlier) until Yolo [87] comes to the stage. Figure 1.2 shows an example of object detection.

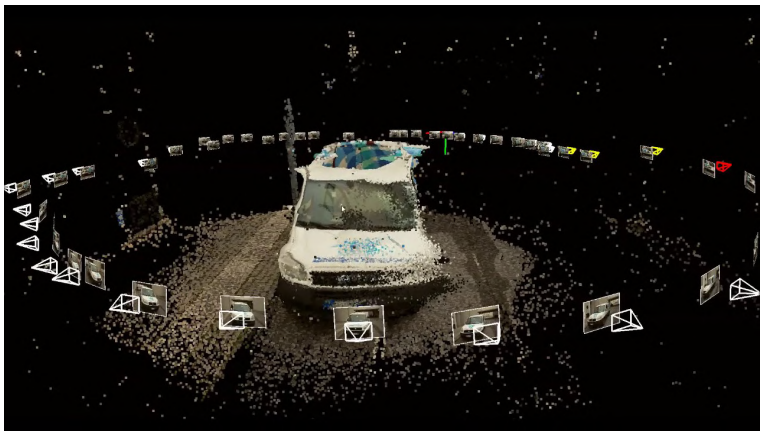


Figure 1.1: Car reconstruction. The white box shows the camera location at different timestamps.

Then after one year, object detection and segmentation go further. Object segmentation means marking every pixel belonging to a specific object. Mask RCNN [51] shows a promising result on this task, as you can see from Figure 1.4. However, Mask RCNN cannot achieve a real-time processing result. Two years later, Yolact [17] reports processing image segmentation 33.5(fps) frames per second. With the mask's help, we can know if a pixel belongs to a particular "potential" dynamic object.

For the algorithm part, we can have a glimpse about what we need to do for 3D reconstruction in a dynamic environment – we need to use depth and RGB image or stereo image to generate 3D points, then we need SLAM to provide the pose estimation to fuse those 3D points so that we can achieve a "global model" for the environment. To deal with the dynamic object, we need to use image segmentation and object detection technique to identify moving objects. From Figure 1.3 we can see the primary research areas related to the problem we want to solve. Based on the

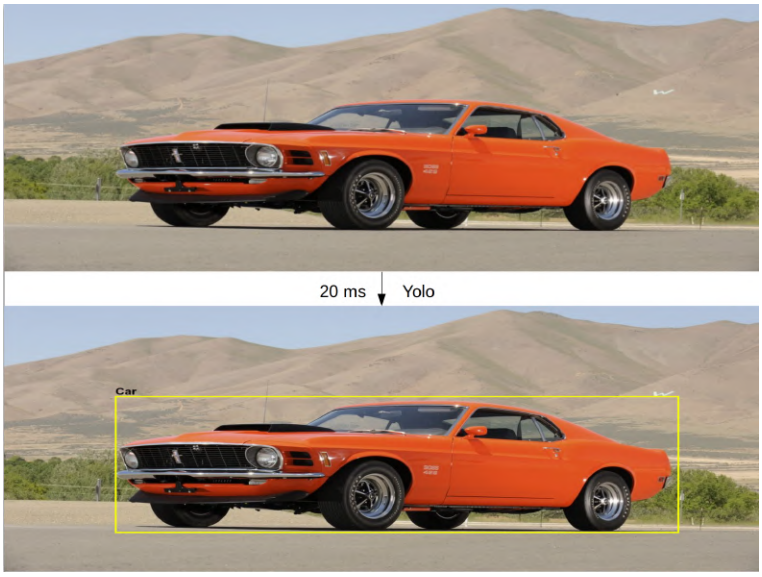


Figure 1.2: Image objet detector Yolo. In 2015 it can use GPU Titan X achieves realtime object detection.

knowledge of those areas, this dissertation has the following contributions:

- Build and test systems that can work on 3D reconstruction even the object has deformation. This system is modulized so people can replace a specific part with minimal work.
- Decouple sparse visual-inertial SLAM and 3D reconstruction system. The former can be used for robust pose estimation, while the latter can focus on fusing new frames.
- The current 3D reconstruction in dynamic environment frequency is lower than 5Hz [9] on CPU. With the help of the "grouped casting" algorithm, which we'll discuss in Chapter 6, we expect more than 10 Hz frequency on CPU. The GPU reconstruction could be real-time on GPU.
- Explore the current state-of-the-art dense mapping algorithm and related work.

The rest of the paper is structured as follows. In chapter 2, 3, 4, 5, we'll talk about the localization and mapping, segmentation, fusion and object tracking theory. In chapter 6, current solutions and my proposed solution is disscussed. The following chapter 6.2.2 shows the current progress. Finally, conclusion chapter 7 will discuss the conclusion and the future work before the final defense.

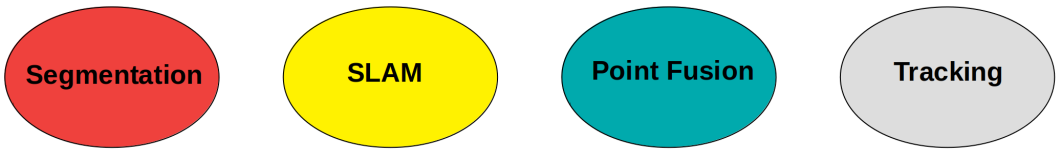


Figure 1.3: Related research areas

Figure 1.4: Image detection and segmentation from Mask RCNN.

Chapter 2

Sparse visual SLAM

In this chapter, we discuss the general idea of the visual SLAM, derive the detailed equation of the visual-inertial preintegration, and present a demo for a tracking system. Sparse visual SLAM's most important task is tracking the sensor (camera or IMU), and most of the SLAM systems are based on the "static world" assumption. Although they are not designed for the dynamic environment, they are normally robust if there aren't too many dynamic objects due to the RANSAC (Random Sample Consensus) algorithm. Some SLAM systems are feature-based [56, 75, 65, 50] and matching the features between the frames is critical, while the others are the so-called "direct" SLAM since they directly use the high gradient pixels as the features [36], [37]. It is inevitable to have some wrong matchings, and they are called "outliers". The RANSAC is designed to reject those outliers. Figure 2.1 shows the difference before and after RANSAC [102]. If the static features are the majority, the wrong matchings and the matching on the dynamic objects will all be seen as outliers, and then they'll not contribute to the camera pose estimation. However, in the real world, we may need to deal with environments where the dynamic objects are the majority. Then it is not enough to reject the dynamic objects just by RANSAC. There are other possible ways to decide if there are potential dynamic objects such as deep learning (directly find the potential moving objects such as people and cars), geometric constraints (dynamic points will violate epipolar constraint) so on and so forth [97, 113, 50, 122]. we conduct some simple experiments by removing the potential dynamic objects in frames with the help of a learning-based segmentation network is presented in Chapter 6.2.2. In our experiment, we use

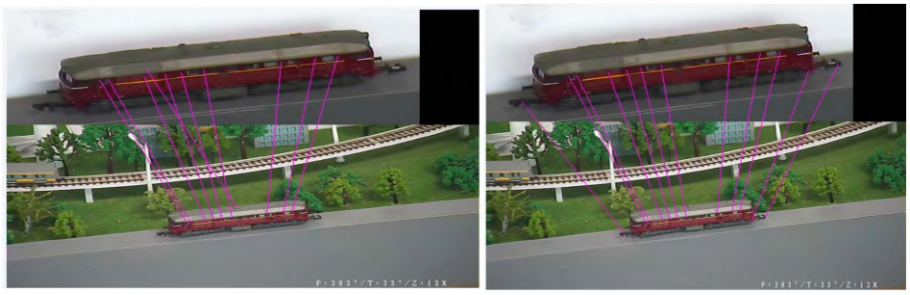


Figure 2.1: We are trying to match the pixels from the same objects's same part. The left side's image has some wrong matchings while the right side's image corrects them according to the RANSAC.

stereo SLAM or stereo-inertial SLAM because the monocular SLAM suffers from scale problems and the monocular-inertial SLAM needs more time for initialization. In the following section, we will briefly go through some important properties for the visual/visual-inertial SLAM. Then we could understand the necessity and tricks applying masks.

2.1 Front End

The "front end" SLAM typically means feature extraction and matching step. During which stage, we try to search robust features that can provide sufficient measurements for the optimization. We are obligated to consider the following requirement when we choose features

- Scale Invariant. We should detect the same features regardless the object's distance w.r.t the camera.
- Rotation Invariant. The feature should be robust to the object rotation.
- Translation Invariant. We are supposed to obtain the same feature when the object have displacement.
- Illumination robustness. In some circumstances, the environment illumination is not consistent. The feature extraction algorithm should be robust to this condition.

- Speed. The feature extraction is just a part of the SLAM system. To make the system in real-time, we have to limit the matching algorithm to a ten-millisecond level.

We should consider the trade-off since we cannot ensure all the properties for a matching algorithm. The classical feature extraction algorithms are SIFT [70], SURF [11], FAST [93], ORB [94], BRIST [64], etc. Various SLAM systems have adopted various feature extraction algorithms. ORBSLAM uses ORB; OKVIS adopts BRIST, for instance. SIFT and SURF are robust to illumination change, translation, and illumination. However, both are computationally expensive, and even the SURF is several times faster than SIFT. They are not suitable for a real-time system for the current CPU. ORB feature achieves a real-time implementation on a low-level CPU while maintaining the invariance requirement. We use the ORB algorithm as an example to illustrate the feature extraction and matching process.

The ORB feature combines FAST feature extraction and BRIEF feature descriptor for matching. The FAST feature extraction has the following steps.

(1) For a pixel p with intensity I_p.

(2) We consider a circle with raduis 3, center p. Each pixel on the circle boundary has an special identity. See Figure 2.2 [93].

(3) Set a threshold t. We check the pixel intensity on the boundary with id 1,5,9,13. If all $I_p - t \geq I_{id}$ or $I_{id} \geq I_p + t$, we investigate if there are more than n contiguous pixels on the circle that have intensity difference larger than t w.r.t the center. We check id 1,5,9,13 firstly because if pair 1,9 or pair 5,13 don't satisfy the requirement, we won't have n contiguous pixels satisfy the t threshold requirement. The n can be 9 or 12. This trival trick accelerate the detection speed.

(4) The neighbor may have several similar features. We use the non-maximal suppression to choose the one that has the most intensity difference.

The FAST detection is not robust to rotation and scale. For this problem, the ORB has the following improvements.

(1) ORB constructs image pyramid and extract features on each pyramid the add the scale invariant property.

(2) The ORB uses "intensity centroid" method to add the rotation invariant property.

The idea of the "intensity centroid" is inspired by the mass centroid. For a rigid body, the vector from the geometry center to the centroid should be constant regardless of the rotation. ORB calculate the intensity centroid of a pattern by the following equation

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y) \tag{2.1}$$

where the order p, q decides the "intensity moment" we want to obtain. $p, q = \{0, 1\}$. x and y are pixel coordinates. $I(x, y)$ is the pixel intensity. $p = 1, q = 0$ gives us the moment accumulation along the x direction, $p = 0, q = 1$ gives us the moment accumulation along the x direction. The pattern orientaion can be given by

$$\theta = \arctan 2(m_{01}, m10) \tag{2.2}$$

With the benefit of the image pyramid and the intensity centroid, the ORB features are more robust to scale and rotation.

ORB uses BRIEF descriptor for feature matching. BRIEF is a binary descriptor. The fundamental idea of the BRIEF is to choose a neighborhood $\mathbf{C}_p$ of a pixel p. In this neighborhood, it randomly chooses n pixel pairs $p_i0, p_i1, i \in [1, n]$ and create binary vector $\mathbf{v}$, length$(\mathbf{v}) = n$. The n is 256 in the ORB. We simply decide the binary element by the pixel intensity.

$$\mathbf{v}[i] = \begin{cases} 1 & I_{p_{i0}} > I_{p_{i1}} \\ 0 & I_{p_{i0}} \leq I_{p_{i1}} \end{cases} \tag{2.3}$$

We use Hamming distance to match the descriptor from distinct features, i.e., accumulate the number of different elements in two binary vectors. The original BRIEF is not robust to the rotation

(note we need the feature rotation robustness and the feature descriptor rotation robustness). The ORB uses "rBRIEF" [94] to boost the rotation invariance. However, the binary descriptor still suffers from a high mismatching rate. The RANSAC is essential to reject the outliers discussed at the beginning of this section.

In the direct SLAM, we don't apply the complex feature extraction and matching algorithm. Instead, we directly pick pixels with high gradient intensities as the features and project them between frames for matching. This brute-force approach leads to a higher mismatching rate. The strategy is to use more pixels to compensate for the mismatched pixels. Thus, the features in the direct SLAM are generally several more times than the feature-based SLAM.

The features concentrate on a portion of an image, which leads to deficient information from the image. An effective strategy to boost the front end performance for tracking is distributing the entire image's feature. Researchers have applied this strategy to either feature-based SLAM [75] or direct SLAM [36]. The ORBSLAM cuts the image into $k * k$ cells and chooses the feature that has the highest quality in each cell. In the direct SLAM, as we need more pixels, we can split the image into numerous $7 * 7$, $9 * 9$ cells can choose the pixel with the highest gradient in each cell. This method results in superior tracking performance.

Based on the matched features, we map pixels from $frameA$ to $frameB$ by the camera motion and projection model. Thus, we can obtain reprojection error and use it as the cost function to optimize the motion model.

2.2 Back End

The "back end" often refers to the step we perform pose optimization or Bundle Adjustment (BA) in the visual SLAM. Bundle Adjustment typically utilizes the conventional Levenberg-Marquardt (LM) or other non-linear least square optimization method. Pose optimization only optimizes the camera motion among frames; BA optimizes the pose and the 2D/3D features.

Besides the general way to perform the LM method, we can use tools such as G2O [61], and Ceres-Solver [1] to conduct back end optimization. Given cost function $\mathbf{e}$ and the jacobian matrix

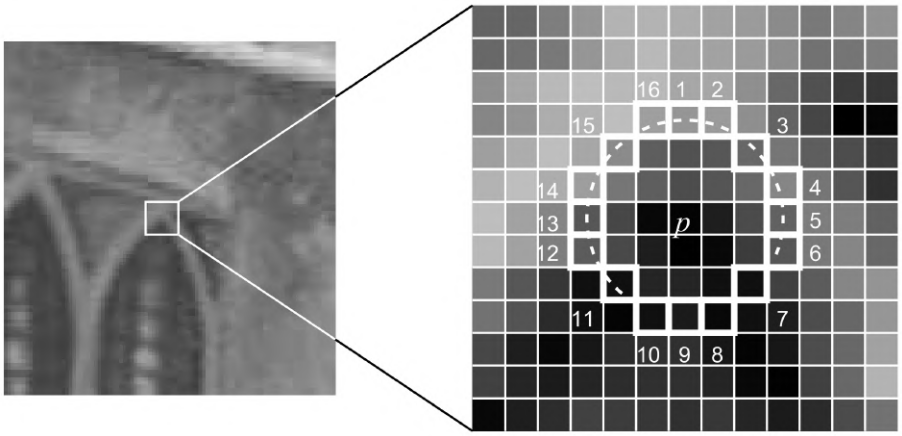


Figure 2.2: FAST feature extraction. We set a threshold t. We check the pixel intensity on the boundary with id 1,5,9,13. If all $I_p - t \geq I_{id}$ or $I_{id} \geq I_p + t$, we investigate if there are more than n contiguous pixels on the circle that have intensity difference larger than t w.r.t the center. We check id 1,5,9,13 firstly because if pair 1,9 or pair 5,13 don't satisfy the requirement, we won't have n contiguous pixels satisfy the t threshold requirement. The n can be 9 or 12.

J w.r.t the target, G2O, and Ceres-Solver can return the target that minimizes the cost function with the LM framework. The target is $\mathbf{T} \in \mathbb{SE}(3)$ for the pose optimization or $\{\mathbf{T}, \mathbf{x}_{k,w}\}$ for the BA, where $\mathbf{x}_{k,w}$ is the 3D landmark k in the world frame w. 3D landmarks are projected from 2D pixels. The $\mathbb{SE}(3)$ is the Special Euclidean Group. It is defined as the following

$$\mathbb{SE}(3) = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \in \mathbb{R}^{4*4} \tag{2.4}$$

where the $\mathbf{t} \in \mathbb{R}^3$ and represents the translation. The $\mathbf{R} \in \mathbb{SO}(3)$ is the Special Orthogonal Group. It is also known as the rotation matrix with the following significant property

$$\begin{aligned} &\mathbf{R} \in \mathbb{R}^{3*3} \\ &\mathbf{R}\mathbf{R}^{\mathrm{T}} = I \\ &\det(\mathbf{R}) = 1 \end{aligned} \tag{2.5}$$

The general pose optimization cost function can be given by the following

$$\mathbf{T}_{\text{best}} = \underset{\mathbf{T}_{i,w}}{\operatorname{argmin}} \sum_{i \in N} \sum_{j \in M} ||\mathbf{e}_{i,j}(\mathbf{T}_{i,w})||_2^2 \tag{2.6}$$

where N is the frame number, and M is the feature number in the world frame. The general BA optimization cost function can be given by the following

$$\{\mathbf{T}_{\text{best}}, \mathbf{x}_{\text{best}}\} = \underset{\mathbf{T}_{i,w}, \mathbf{x}_{k,w}}{\operatorname{argmin}} \sum_{i \in N} \sum_{j \in M} ||\mathbf{e}_{i,j}(\mathbf{T}_{i,w}, \mathbf{x}_{k,w})||_2^2 \tag{2.7}$$

Note the k is the 3D landmark id in the world frame. We assign a new landmark id once we project a new feature from the pixel to the world point according to the camera model and transformation matrix $\mathbf{T}$. The above equation doesn't apply to every SLAM implementation. For example, the DSO [36] doesn't optimize the 3D landmark but the 1D depth of the pixel. Thus, our BA optimization becomes to

$$\{\mathbf{T}_{\text{best}}, d_{i,j}\} = \underset{\mathbf{T}_{i,w}, d_{i,j}}{\operatorname{argmin}} \sum_{i \in N} \sum_{j \in M} ||\mathbf{e}_{i,j}(\mathbf{T}_{i,w}, d_{i,j})||_2^2 \tag{2.8}$$

At the same time, different SLAM implementations may add more parameters into optimization, such as IMU bias, illumination parameter, etc. However, the above equation gives us a glimpse of

what the "back end" typically works. The difference between the pure pose optimization and BA is what we want to add to our constraints and what we want to optimize.

Given the optimization goal, we want to specify the detail of the error function. The camera observations are 2D pixels $\mathbf{z}_{i,j}$, we project them into the 3D world to obtain the 3D landmarks. Then for the error function, we can have 3D-3D matching (landmark position-landmark position), 2D-3D matching (pixel position-landmark position), 2D-2D matching (pixel position-pixel position), or 1D-1D matching (pixel intensity-pixel intensity) depending on the application.

$$\mathbf{e}_{i,j} = \begin{cases} (\pi(\mathbf{z}_{i,j}), \mathbf{T}_{i,w}) - \mathbf{x}_{k,w} \quad 3\mathrm{D}-3\mathrm{Dmatching} \\ \mathbf{z}_{i,j} - \pi(\mathbf{T}_{i,w}, \mathbf{x}_{k,w}) \quad 2\mathrm{D}-3\mathrm{Dmatching} \\ \mathbf{z}_{i,j} - \pi(\mathbf{T}_{i,w}, \mathbf{T}_{ref,w}, \pi^{-1}(\mathbf{z}_{i,j})) \quad 2\mathrm{D}-2\mathrm{Dmatching} \\ I(\mathbf{z}_{i,j}) - I(\pi(\mathbf{T}_{w}, \mathbf{T}_{ref,w}, \pi^{-1}(\mathbf{z}_{i,j}))) \quad 1\mathrm{D}-1\mathrm{Dmatching} \end{cases} \tag{2.9}$$

where $\mathbf{T}_{ref,w}$ is the pose in the reference frame, $\mathbf{T}_{ref,w}$ remains unchanged during the optimization. Those equations are essentially the same but use different criteria. The ORBSLAM uses 2D-3D matching while the DSO uses 1D-1D intensity matching, for example. Note in the above equation, we may have two $\mathbf{z}_{i,j}$. Note the "j" here stands for the global map point id, so the two $\mathbf{z}$ stands for one global map point's projection in two frames, which has a bit of symbol abuse here. This mapping is more clear in Eq. (2.10).

Given the cost function, we can derive the Jacobian matrix of the cost function. Here we don't intend to derive Jacobians for all of the cost functions. We use the 1D-1D matching pose optimization as an example for the following section. We let $E = \sum_{i \in N} \sum_{j \in M} ||\mathbf{e}_{i,j}(\mathbf{T}_{i,w})||_2^2$. In pose optimization, we generally use only two frames for a fast LM converge. One reference frame with pose $\mathbf{T}_{ref,w}$ and one target frame with pose $\mathbf{T}_w$ for optimization. We can omit frame id i.

Then we can simplify the cost function as

$$\begin{aligned}\mathbf{e} &= I(\mathbf{z}_{j0}) - I(\pi(\mathbf{T}_w, \mathbf{T}_{ref,w}, \pi^{-1}(\mathbf{z}_j))) \\ &= I(\mathbf{z}_{j0}) - I(\pi(\mathbf{T}_w, \mathbf{x}_{j,w})) \\ &= I(\mathbf{z}_{j0}) - I(\pi(\mathbf{x}_{j,c})) \\ &= I(\mathbf{z}_{j0}) - I(\mathbf{z}_{j1})\end{aligned} \tag{2.10}$$

where $\mathbf{x}_{j,w}$ is the $\mathbf{z}_{j1}$ corresponding landmark in the world frame, $\mathbf{x}_{j,c}$ is the $\mathbf{x}_{j,w}$ in the target camera frame, $\mathbf{z}_{j1}$ in the pixel position in the target frame projected by the $\mathbf{x}_{j,c}$. The Jacobian matrix is $\partial\mathbf{e}/\partial\mathbf{T}_w$. For simplification, we write $\mathbf{x}_{j,n}$ as $\mathbf{x}$. Apply the chain rule, we have

$$\mathbf{J} = \frac{\partial \mathbf{e}}{\partial \mathbf{T}_w} = -\frac{\partial I(\mathbf{z}_{j1})}{\partial \mathbf{z}_{j1}} \frac{\partial \mathbf{z}_{j1}}{\partial \mathbf{p}_{dis}} \frac{\partial \mathbf{p}_{dis}}{\partial \mathbf{p}_{undis}} \frac{\partial \mathbf{p}_{undis}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \mathbf{T}_w} \tag{2.11}$$

where $\mathbf{p}_{dis}$ is the 2D point on the camera plane, $\mathbf{p}_{undis}$ is the $\mathbf{p}_{dis}$ on the camera plane considering the distortion.

For the $\frac{\partial I(\mathbf{z}_{j1})}{\partial \mathbf{z}_{j1}}$, we simply use the pixel gradient vertically and horizentally. We use u, v to denote the $\mathbf{z}_{j1}$ position on the image. i.e. $\mathbf{z}_{j1} = [u, v]^T$.

$$\begin{aligned}\frac{\partial I(\mathbf{z}_{j1,u,v})}{\partial \mathbf{z}_{j1}} &= [\nabla I_x(\mathbf{z}_{j1,u,v}), \nabla I_y(\mathbf{z}_{j1,u,v})] \\ \nabla I_x(\mathbf{z}_{j1,u,v}) &= (I(\mathbf{z}_{j1,u+1,v}) - I(\mathbf{z}_{j1,u-1,v}))/2 \\ \nabla I_y(\mathbf{z}_{j1,u,v}) &= (I(\mathbf{z}_{j1,u,v+1}) - I(\mathbf{z}_{j1,u,v-1}))/2\end{aligned} \tag{2.12}$$

$\frac{\partial \mathbf{z}_{j1}}{\partial \mathbf{p}}$ is from the camera projection function. Assume we use pinhole model then we have

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \end{bmatrix} \begin{bmatrix} x_{dis} \\ y_{dis} \end{bmatrix} \tag{2.13}$$

where $\mathbf{p}_{dis} = [x_{dis}, y_{dis}, 1]^T$, $[f_x, f_y, c_x, c_y]$ are the camera focal length and camera center. Then we have

$$\frac{\partial \mathbf{z}_{j1}}{\partial \mathbf{p}_{dis}} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \end{bmatrix} \tag{2.14}$$

From $\mathbf{p}_{dis}$ to $\mathbf{p}_{undis} = [x_{undis}, y_{undis}]^T$ we undistort the camera. For simplification we let $\mathbf{p}_{undis} = [x, y]^T$. Assume we use radial-tangential distortion model. We have

$$\begin{aligned} x_{dis} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 xy + p_2(r^2 + 2x^2) \\ y_{dis} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_2 xy + p_1(r^2 + 2y^2) \end{aligned} \tag{2.15}$$

where $r^2 = x^2 + y^2$, k_1, k_2, k_3, p_1, p_2 are the distortion paramters. Based on the distortion model, we can have the partial derivative

$$\begin{aligned} \frac{\partial \mathbf{p}_{dis}}{\partial \mathbf{p}_{undis}} &= \begin{bmatrix} \frac{\partial x_{dis}}{x} & \frac{\partial x_{dis}}{\partial y} \\ \frac{\partial y_{dis}}{\partial x} & \frac{\partial y_{dis}}{\partial y} \end{bmatrix} \\ \frac{\partial x_{dis}}{x} &= 1 + k_1(r^2 + 2x^2) + k_2(r^4 + 4r^2x^2) + k_3(r^6 + 6r^4x^2) + 2p_1 y + 6p_2 x \\ \frac{\partial x_{dis}}{\partial y} &= 2k_1 xy + 4k_2 xyr^2 + 6r^4 k_3 xy + 2p_1 x + 2p_2 y \\ \frac{\partial y_{dis}}{\partial x} &= \frac{\partial x_{dis}}{\partial y} \\ \frac{\partial y_{dis}}{\partial y} &= 1 + k_1(r^2 + 2y^2) + k_2(r^4 + 4r^2y^2) + k_3(r^6 + 6r^4y^2) + 2p_1 x + 6p_2 y \end{aligned} \tag{2.16}$$

$\mathbf{p}_{undis}$ is the 2D projection of the 3D point $\mathbf{x} = [X, Y, Z]$ on the camera frame.

$$\begin{aligned} x &= X/Z \\ y &= Y/Z \end{aligned} \tag{2.17}$$

The we can have the jacobian matrix as

$$\frac{\partial \mathbf{p}_{undis}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial x}{\partial X} & \frac{\partial x}{\partial Y} & \frac{\partial x}{\partial Z} \\ \frac{\partial y}{\partial X} & \frac{\partial y}{\partial Y} & \frac{\partial y}{\partial Z} \end{bmatrix} = \begin{bmatrix} \frac{1}{Z} & 0 & -\frac{X}{Z^2} \\ 0 & \frac{1}{Z} & -\frac{Y}{Z^2} \end{bmatrix} \tag{2.18}$$

$\frac{\partial \mathbf{x}}{\partial \mathbf{T}_w}$ is not straightforward comparing to others. Since $\mathbf{T}_w \in \mathbb{SE}(3)$, we cannot apply the conventional method to obtain the derivatives to it. Why? We can recapture the definition of the derivative given a function $f(x)$ with scalar domain x.

$$f'(x) = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \tag{2.19}$$

where h is very small and we can call it a "disturbance". To apply the above equation, the domain x must have the closure of summation. $x \in \mathbb{R}$ has the closure of summation. However, the

$\mathbb{SE}(3)$ doesn't have this property. The closure of the summation has definition on the Lie Algebra $\boldsymbol{\xi} = [\boldsymbol{\theta}, \mathbf{t}] \in \mathfrak{se}(3)$. $\mathfrak{se}(3) =\in \mathbb{R}^6$, $\mathbf{t} \in \mathbb{R}^3$, $\mathfrak{so}(3) \in \mathbb{R}^3$ is the lie algebra of the $\mathbb{SO}(3)$. The first three elements represent the orientation and the last three elements represent the translation. We define the "hat" operation for a vector $\mathbf{m} = [m_1, m_2, m_3]$ as

$$\mathbf{m}^\wedge = \mathbf{M} = \begin{bmatrix} 0 & -m_3 & m_2 \\ m_3 & 0 & -m_1 \\ -m_2 & m_1 & 0 \end{bmatrix} \tag{2.20}$$

We have the mapping between Lie Group and its Lie Algebra

$$\begin{aligned} &\exp(\boldsymbol{\theta}^\wedge) = \mathbf{R} \\ &\log(\mathbf{R})^\vee = \boldsymbol{\theta} \\ &\exp(\boldsymbol{\xi}^\wedge) = \begin{bmatrix} \exp(\boldsymbol{\theta}^\wedge) & \mathbf{Jt} \\ \mathbf{0} & 1 \end{bmatrix} = \mathbf{T} \\ &\log(\mathbf{T})^\vee = \boldsymbol{\xi} \\ &\mathbf{a}^\wedge \mathbf{b} = -\mathbf{b}^\wedge \mathbf{a} \end{aligned} \tag{2.21}$$

The sepcial "log","exp" operation is in [43]. The most significant feature we need to notice is using the conventional method for derivation on the Lie Algebra. We have some powerful properties for the above mapping [105].

$$\begin{aligned} &\exp(\boldsymbol{\xi}^\wedge) \approx \mathbf{I} + \boldsymbol{\xi} \quad \text{if} \quad \boldsymbol{\xi} \to \mathbf{0} \\ &\exp(\boldsymbol{\theta}^\wedge) \approx \mathbf{I} + \boldsymbol{\theta} \quad \text{if} \quad \boldsymbol{\theta} \to \mathbf{0} \\ &\exp((\boldsymbol{\theta} + \boldsymbol{\delta\theta})^\wedge) \approx \exp(\boldsymbol{\theta}^\wedge)\exp(J_r(\delta\boldsymbol{\theta})\delta\boldsymbol{\theta}^\wedge) \\ &\exp(\boldsymbol{\theta}^\wedge)\mathbf{R} = \mathbf{R}\exp((\mathbf{R}^T\boldsymbol{\theta})^\wedge) \\ &\log(\exp(\boldsymbol{\theta}_0^\wedge)\exp(\boldsymbol{\theta}_1^\wedge))^\vee \approx \begin{cases} J_l(\boldsymbol{\theta}_1)^{-1}\boldsymbol{\theta}_0 + \boldsymbol{\theta}_1 & \text{if} \quad \boldsymbol{\theta}_0 \to \mathbf{0} \\ J_r(\boldsymbol{\theta}_0)^{-1}\boldsymbol{\theta}_1 + \boldsymbol{\theta}_0 & \text{if} \quad \boldsymbol{\theta}_1 \to \mathbf{0} \end{cases} \end{aligned} \tag{2.22}$$

$J_r(\boldsymbol{\theta}) = J_l(-\boldsymbol{\theta})$. The $J_r(\boldsymbol{\theta})$ is the right Jacobian of the $\mathbb{SO}(3)$ [26, p.40].

$$J_r(\boldsymbol{\theta}) = \mathbf{I} - \frac{1 - \cos(||\boldsymbol{\theta}||)}{||\boldsymbol{\theta}||^2}\boldsymbol{\theta}^\wedge + \frac{1 - \sin(||\boldsymbol{\theta}||)}{||\boldsymbol{\theta}||^3}(\boldsymbol{\theta}^\wedge)^2 \tag{2.23}$$

We can conduct various simplification using the above properties. For example, if we have a continuous multiplication of $\delta\boldsymbol{\theta}_k$ with exponential mapping and all $\delta\boldsymbol{\theta}_k \to \mathbf{0}$. We let a combined exponential mapping $\exp(\delta\boldsymbol{\phi}_i^\wedge)$ to descripe the production result. i.e, $\exp(\delta\boldsymbol{\phi}_i^\wedge) \doteq \prod_{k=0}^{i} \exp(\delta\boldsymbol{\theta}_k^\wedge)$. For the same reasoning, we use $\exp(\delta\boldsymbol{\phi}_{i-1}^\wedge)$ to describe the combined exponential map from $k=1$ to i. i.e. $\exp(\delta\boldsymbol{\phi}_{i-1}) \doteq \prod_{k=1}^{i} \exp(\delta\boldsymbol{\theta}_k^\wedge)$. We apply the same definition to $\{\delta\boldsymbol{\phi}_{i-2}\ ...\ \delta\boldsymbol{\phi}_0\}$. We can infer the $\delta\boldsymbol{\phi}_0 = \delta\boldsymbol{\theta}_0$ according to the above definition. If we apply the logrithm to the $\delta\boldsymbol{\phi}$, we can have the following approximation

$$
\begin{aligned}
\delta\boldsymbol{\phi}_i = \log(\prod_{k=0}^{i} \exp(\delta\boldsymbol{\theta}_k^\wedge))^\vee &\overset{(1)}{=} \log(\exp(\delta\boldsymbol{\theta}_0^\wedge)\exp(\delta\boldsymbol{\theta}_1^\wedge)\exp(\delta\boldsymbol{\theta}_2^\wedge)...)^\vee \\
&\overset{(2)}{\doteq} \log(\exp(\delta\boldsymbol{\theta}_0^\wedge)\exp(\delta\boldsymbol{\phi}_{i-1}^\wedge))^\vee \\
&\overset{(3)}{\approx} J_r(\delta\boldsymbol{\theta}_0)^{-1}\delta\boldsymbol{\phi}_{i-1} + \delta\boldsymbol{\theta}_0 \\
&\overset{(4)}{\approx} \delta\boldsymbol{\phi}_{i-1} + \delta\boldsymbol{\theta}_0
\end{aligned}
$$

$$
\begin{aligned}
\delta\boldsymbol{\phi}_{i-1} = \log(\prod_{k=1}^{i} \exp(\delta\boldsymbol{\theta}_k^\wedge))^\vee &= \log(\exp(\delta\boldsymbol{\theta}_1^\wedge)\exp(\delta\boldsymbol{\theta}_2^\wedge)\exp(\delta\boldsymbol{\theta}_3^\wedge)...)^\vee \\
&= \log(\exp(\delta\boldsymbol{\theta}_0^\wedge)\exp(\delta\boldsymbol{\phi}_{i-2}^\wedge))^\vee \\
&\approx J_r(\delta\boldsymbol{\theta}_0)^{-1}\delta\boldsymbol{\phi}_{i-2} + \delta\boldsymbol{\theta}_0 \\
&\approx \delta\boldsymbol{\phi}_{i-2} + \delta\boldsymbol{\theta}_1 \\
&... \\
\delta\boldsymbol{\phi}_1 = \log(\exp(\delta\boldsymbol{\theta}_{i-1}^\wedge)\exp(\delta\boldsymbol{\theta}_i^\wedge))^\vee &\approx \delta\boldsymbol{\theta}_{i-1} + \delta\boldsymbol{\theta}_i
\end{aligned}
\tag{2.24}
$$

From the subequation (1) to (2), we use the definition of the $\delta\boldsymbol{\phi}$. From (2) to (3), we use the first logorithm property in Eq. (2.22). Both $\delta\boldsymbol{\phi}$ and $\delta\boldsymbol{\theta}$ are small numbers. From (3) to (4), we consider the Eq. (2.23). When the $\boldsymbol{\theta}$ is small, the jacobian is approximately equal to identity matrix. We can then apply the same reasoning to the $\{\delta\boldsymbol{\phi}_{i-1}, \delta\boldsymbol{\phi}_{i-2}\ ...\ \delta\boldsymbol{\phi}_1\}$. Bring the result of $\delta\boldsymbol{\phi}_1$ back to $\delta\boldsymbol{\phi}_2$ then bring the result of $\delta\boldsymbol{\phi}_2$ to $\delta\boldsymbol{\phi}_3$ continuously, we finally can have $\delta\boldsymbol{\phi} \approx \delta\boldsymbol{\theta}_0 + \delta\boldsymbol{\theta}_1 + ... + \delta\boldsymbol{\theta}_i$.

Then we have

$$\delta\boldsymbol{\phi}_i = \sum_{k=0}^{i} \delta\boldsymbol{\theta}_k \approx \log(\prod_{k=0}^{i} \exp(\delta\boldsymbol{\theta}_k^{\wedge}))^{\vee} \Longrightarrow \exp((\sum_{k=0}^{i} \delta\boldsymbol{\theta}_k)^{\wedge}) \approx \prod_{k=0}^{i} \exp(\delta\boldsymbol{\theta}_k^{\wedge}) \tag{2.25}$$

The above equation converts the continuous exponential mapping to summation magically. We'll use this approximation in the visual inertial odometry introduction.

Now we infer the property of the basic derivative of the Lie Group based on the above properties. $\frac{\partial \mathbf{x}}{\partial \mathbf{T}_w}$ can be written as its partial derivative to the Lie Algebra

$$\frac{\partial \mathbf{x}}{\partial \mathbf{T}_w} = \frac{\partial \mathbf{T}_w \mathbf{x}_w}{\partial \mathbf{T}_w} \rightarrow \frac{\partial \exp(\boldsymbol{\xi}^{\wedge})\mathbf{x}_w}{\partial \boldsymbol{\xi}} \tag{2.26}$$

Now we can add a disturbance to the $\boldsymbol{\xi}$. There are two ways to derive this. The first way is to add disturbance directly on the Lie Algebra since it has the closure of summation.

$$\frac{\partial \exp(\boldsymbol{\xi}^{\wedge})\mathbf{x}_w}{\partial \boldsymbol{\xi}} = \lim_{\delta\boldsymbol{\xi}\rightarrow\mathbf{0}} \frac{\exp((\boldsymbol{\xi}+\delta\boldsymbol{\xi})^{\wedge})\mathbf{x}_w - \exp(\boldsymbol{\xi}^{\wedge})\mathbf{x}_w}{\delta\boldsymbol{\xi}} \tag{2.27}$$

However, in this way, the result is more complex than the second approach since it requires the "left jacobian" estimation [105]. When we have numerous error terms, we need the Jacobian matrix calculation as cheap as possible. It is more general to use the second approach, which adds disturbance to the transformation matrix. We use multiplication instead of "add" for the disturbance. To understand this, we need to recapture the idea behind the derivation: we should "add" a "small" number to the domain that leads to a "small" change on the codomain, and we calculate the change rate. For a $\mathbb{SE}(3)$ transformation matrix, if we want to "add" anything to the transformation matrix $\mathbf{T}_1$, it results in moving it to $\mathbf{T}_2 = \mathbf{T}\mathbf{T}_1$. Then the way we calculate the "movement" is $\mathbf{T} = \mathbf{T}_2\mathbf{T}_1^{-1}$. Then a straightforward illustration for the second approach is any "disturbance" applying to the $\mathbb{SE}(3)$ result in multiplication. the $\mathbb{SE}(3)$ has a closure of

multiplication. Thus, we can "add" a disturbance to the $\mathbf{T}_w$. Let $\delta\boldsymbol{\upsilon} = [\delta\mathbf{x}, \delta\boldsymbol{\theta}]$

$$\begin{aligned}
\frac{\partial \exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w}{\partial \boldsymbol{\xi}} &= \lim_{\delta\boldsymbol{\upsilon}\to\mathbf{0}} \frac{\exp(\delta\boldsymbol{\upsilon}^\wedge)\exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w - \exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w}{\delta\boldsymbol{\upsilon}} \\
&\overset{Eq.(2.22)}{\approx} \lim_{\delta\boldsymbol{\upsilon}\to\mathbf{0}} \frac{(\mathbf{I}+\delta\boldsymbol{\upsilon}^\wedge)\exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w - \exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w}{\delta\boldsymbol{\upsilon}} \\
&= \lim_{\delta\boldsymbol{\upsilon}\to\mathbf{0}} \frac{\delta\boldsymbol{\upsilon}^\wedge \exp(\boldsymbol{\xi}^\wedge)\mathbf{x}_w}{\delta\boldsymbol{\upsilon}} \\
&= \lim_{\delta\boldsymbol{\upsilon}\to\mathbf{0}} \frac{\begin{bmatrix} \delta\boldsymbol{\theta}^\wedge & \delta\mathbf{x} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R}\mathbf{x}_w + \mathbf{t} \\ 1 \end{bmatrix}}{\delta\boldsymbol{\upsilon}} \\
&= \lim_{[\delta\mathbf{x},\delta\boldsymbol{\theta}]\to\mathbf{0}} \frac{\begin{bmatrix} \delta\boldsymbol{\theta}^\wedge(\mathbf{R}\mathbf{x}_w + \mathbf{t}) + \delta\mathbf{x} \\ 1 \end{bmatrix}}{[\delta\mathbf{x}, \delta\boldsymbol{\theta}]} \\
&= \begin{bmatrix} \mathbf{I} & -(\mathbf{R}\mathbf{x}_w + \mathbf{t}) \\ \mathbf{0} & \mathbf{0} \end{bmatrix}
\end{aligned} \tag{2.28}$$

Combine the results of Eq. (2.12), (2.14), (2.18), (2.28) we can have the Jacobian matrix in Eq. (2.11). Given Eq. (2.11) and Eq. (2.10), we can use the G2O or Ceres-Solver to find the target that can minimize the cost function. Behind the scene, they use the Gaussin-Newton/Levenberg–Marquardt algorithm with the Jacobian matrix. Given the cost function $||f(x)||_2^2$ and a prior x_0, the Gaussian-Newton updates a δx that leads the cost to a local minimum.

$$\mathbf{J}^T\mathbf{J}\delta x = -\mathbf{J}^T f(x) \tag{2.29}$$

The above equation is the well-known Normal Equation. It updates the δx in each iteration and then updates the $x_{k+1} = x_k + \delta x$, k is the iteration step. For the pose estimation in our implementation, we have the Jacobian matrix given by Eq. (2.11) and the cost given by Eq. (2.10). The normal equation is also written as

$$\begin{aligned}
&\mathbf{H}\delta x = \boldsymbol{g} \\
&\boldsymbol{g} := -\mathbf{J}^T f(x) \\
&\mathbf{H} := \mathbf{J}^T\mathbf{J}
\end{aligned} \tag{2.30}$$

Gaussian Newton converges fast, but it may lead to a more significant error. The gradient descent algorithm can relieve that problem but converges slowly near to the minimum. The Levenberg–Marquardt updates the δx but adds a step control parameter λ to combine Gaussian Newton and gradient descent algorithm.

$$(\mathbf{H} + \lambda \mathbf{I})\delta x = \boldsymbol{g} \tag{2.31}$$

The λ is customized and a weight parameter.If the δx leads to a smaller error in the next iteration, we let $\lambda = \lambda/10$, otherwise we let $\lambda = \lambda * 10$. This operation gives us a Gaussian-Newton algorithm when the λ is small as the $\mathbf{H}$ has more weights. It gives us a gradient descent algorithm when the λ is large because the identity matrix weighs more. Researchers have discussed the derivation of the Gaussian-Newton/Levenberg–Marquardt in the SLAM application in detail [48], [106].

2.2.1 Implementation for Pose Optimization

This section will show a basic implementation of the above "front end" and "back end." We'll use the ORBSLAM2 as the framework with modifications. In the "front end", we use pixels with high gradient intensities to replace the ORB features so that we can use cost function Eq. (2.10). i.e., the "direct" approach. In the "back end", we use G2O to implement the Levenberg–Marquardt algorithm.

In the front end we just choose high gradient pixels in each 7 by 7 pattern. We'll have a brief discussion about the back end implementation. We discussed before we need to input two significant terms to the G2O: cost function and Jacobians. We need to define the cost function in the *computeError()* function and the Jacobians in the *linearizeOplus()* function. The *computeError()* correspondes to Eq. 2.10.

```
virtual void computeError()
{
    const VertexSE3Expmap* v  =static_cast<const VertexSE3Expmap*> ( _vertices
[0] );
    Eigen::Vector3d x_local = v->estimate().map ( x_world_ );
    double xx = x_local[0]/x_local[2];
```

```
        double yy = x_local[1]/x_local[2];
        double x,y;
        distort(xx,yy,x,y);
        ...
        _error ( 0,0 ) = get_interpolated_pixel_value ( x,y ) - _measurement;
    }
```

where the *x_world_* correspondes to the $\mathbf{x}_{j,w}$ in Eq. (2.10), the *map* indicates applying the transformation $\mathbf{T}_w$, the reprojection from the *x_local_* to the xx, yy and the *distort* function correspondes to the $\pi(\mathbf{x}_{j,c})$. We get the reprojection pixel location. The pixel location is *float* type. Thus, we use bilinear interpolation to acquire its intensity. Finally, we subtract two intensities.

For the Jacobian function, we apply it according to the chain rules in Eq. (2.11). We define the following variables

```
void EdgeSE3ProjectDirectOnlyPose::linearizeOplus(){
    ...
    Eigen::Matrix<double, 1, 2> jacobian_pixel_uv;
    Eigen::Matrix2d jacobian_uv_cam;
    Eigen::Matrix2d jacobian_cam_dist;
    Eigen::Matrix<double,2,3> jacobian_dist_pc;
    Eigen::Matrix<double,3,6> jacobian_pc_ksai;

    Eigen::Matrix<double, 2, 6> jacobian_uv_ksai;
    jacobian_uv_ksai = jacobian_uv_cam * jacobian_cam_dist * jacobian_dist_pc *
   jacobian_pc_ksai;
    _jacobianOplusXi = jacobian_pixel_uv*jacobian_uv_ksai;
    ...
}
```

whre the *jacobian_pixel_uv*, *jacobian_uv_cam*, *jacobian_cam_dist*, *jacobian_dist_pc*, *jacobian_pc_ksai* correspondes to the $\frac{\partial I(\mathbf{z}_{j1})}{\partial \mathbf{z}_{j1}}$, $\frac{\partial \mathbf{z}_{j1}}{\partial \mathbf{p}_{dis}}$, $\frac{\partial \mathbf{p}_{dis}}{\partial \mathbf{p}_{undis}}$, $\frac{\partial \mathbf{p}_{undis}}{\partial \mathbf{x}}$, $\frac{\partial \mathbf{x}}{\partial \mathbf{T}_w}$ in Eq. (2.11) respectively. *_jacobianOplusXi* is the final Jacobian matrix we want to obtain.

Fig. 2.3 shows the difference between the ORB features and the high gradient pixels, i.e.,

direct features. The ORB features are more sparse than the direct features. We typically choose only hundreds of ORB features but thousands of high gradient pixels. We have to increase the "quantity" to achieve the "quality" for the direct method since the direct feature is easier to have a false matching. We try to choose a feature in each 7 by 7 or 9 by 9 pattern. In this way, the high gradient pixels spread out to the whole image. This approach can avoid the pixels concentrating at a local part of the image and falling into a local minimum optimization result. It is very straightforward to parallel the process of finding the high gradient pixels. For a 640 by 480 image, we need around $640 * 480/(7 * 7) \approx 6269$ parallel threads, which is easy to run on the modern GPU. For this reason, we run the high gradient pixel detection on the GPU with CUDA.

We use the TUM RGB-D [108] dataset to test the algorithm. Fig. 2.4 shows the tracking

Figure 2.3: The left side of the image shows the result of the original ORBSLAM2 feature. The right side of the image shows the effect of high gradient pixels. i.e. "direct features". The ORB features are more sparse than the direct features. Instead of choosing features with gradients larger than a threshold, we choose features in each 7 by 7 or 9 by 9 pattern. In this way, the high gradient pixels spread out to the whole image.

result on the TUM RGB-D fr1-teddy dataset. The blue frames result from the original ORBSLAM, and the pink frames result from the direct method. Note that the ORBSLAM only shows the "keyframe" in their map. Although our test implementation doesn't have a "keyframe" but does frame-by-frame tracking, we show the frame with the same id as the ORBSLAM keyframe. In this

way, we can be more clear about how the direct method follows the ORBSLAM. From the left column to the right column, we have more frames in the localization system. The direct method follows the ORBSLAM initially but has a larger drift as we have more frames, which is expected. In our algorithm, we optimize the pose frame by frame, but the ORBSLAM updates the pose and the map. The pure pose optimization accumulates error faster than the visual odometry with bundle adjustment.

Fig. 2.5 shows another example using TUM RGB-D fr1-desk dataset. In this dataset, there is a rapid turn, leading to failure tracking with the direct method. We show this situation in the right column of the figure. One of the shortcomings of the direct method is that it is not as robust as the feature-based method w.r.t. a rapid movement. Another shortcoming is it is not robust to illumination change [82]. The direct method assumes the illumination is consistent.

2.2.2 Visual Bundle Adjustment

In the above section we conduct the implementation of the pose optimization, which is the "localization" part of the SLAM. The core of the "back end", however, is the Bundle Adjustment (BA), which optimize the pose as well as the landmarks (3D word points or point depth). The cost function is Eq. (2.8) while the Jacobian matrix is different. For a set of frames $i \in N$ and the features $j \in M$, we can write the Jacobian matrix structure as the Eq. (2.34). where $\mathbf{A}_{i,j} = \frac{\partial \mathbf{e}_{i,j}}{\partial \mathbf{T}_{i,w}}$ is the jacobian matrix of the cost w.r.t. frame i pose, $\mathbf{B}_{i,j} = \frac{\partial \mathbf{e}_{i,j}}{\partial \mathbf{x}_{j,w}}$ is the jacobian matrix of the cost w.r.t. feature j. The BA has the following structure because we might be able to see a feature in multiple frames. Thus, each feature contributes to several frames pose optimization, which is the $\mathbf{A}$ matrix. At the same time, several frames also contribute to one feature's oprimization, which is the $\mathbf{B}$ matrix. This is shown in Fig. 2.6 [21]. From Eq. (2.34) structure we can extract two main blocks.

Figure 2.4: TUM RGB-D fr1-teddy dataset tracking result. The blue frames result from the original ORBSLAM, and the pink frames result from the direct method. The direct method follows the ORBSLAM initially but has a more considerable drift as we have more and more frames.

Figure 2.5: TUM RGB-D fr1-desk dataset tracking result. The pink frames are from the direct method, while the blue frames are from the ORBSLAM. There is a rapid turn in the TUM RGB-D fr1-desk dataset, which leads to failure tracking. See the right column

$$\mathbb{A}_j = \overbrace{\begin{bmatrix} \mathbf{A}_{0j} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{1j} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_{nj} \end{bmatrix}}^{0\ldots n} \quad \mathbb{B}_j = \overbrace{\begin{bmatrix} \mathbf{0} & \cdots & \mathbf{B}_{0j} & \cdots & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{B}_{1j} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \cdots & \mathbf{B}_{nj} & \cdots & \mathbf{0} \end{bmatrix}}^{0\ldots j \quad j\ldots m} \tag{2.32}$$

Then our final matrix structure of the Eq. (2.34) can be simplified as Eq. (2.33). We decide the dimension of the matrix by the number of map points M, the number of frames N, the dimension of the cost function $\alpha - \beta$, where the $\alpha = 2,\ \beta = 3$ for 2D-3D matching. For 1D-1D matching from Eq. (2.10), the $\mathbf{A}_{ij}$ has dimension 1 by 6. We have derived its detail in Eq 2.11. The DSO only optimize the map point depth when it applies 1D-1D matching cost so the $\mathbf{B}_{ij}$ has dimension 1 by 1. However, for the 2D-3D matching like the ORBSLAM, the $\mathbf{A}_{ij}$ has dimension 2 by 6 while the $\mathbf{B}_{ij}$ is 2 by 3 since ORBSLAM optimizes 3D landmarks. The total dimension of the $\mathbf{J}$ is $col * row$, where the column number is $col = (6N + \beta M)$ and the row number is $row = \alpha NM$. Typically, the feature number can be thousands. Due to the large size of the Jacobian matrix, it is expensive to apply the LM algorithm because we need to inverse the $\mathbf{H}$ in Eq. (2.31). However, from Eq. (2.34) we can know the matrix is sparse, furthermore, the $\mathbb{A}$ is a block-diagonal matrix. Given this property, researchers apply the Schur complement to reduce the computation complexity [103].

$$\mathbf{J} = \begin{bmatrix} \mathbb{A}_0 & \mathbb{B}_0 \\ \mathbb{A}_1 & \mathbb{B}_1 \\ \vdots & \vdots \\ \mathbb{A}_m & \mathbb{B}_m \end{bmatrix} \tag{2.33}$$

$$\mathbf{J} = \begin{bmatrix} \mathbf{A}_{00} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{B}_{00} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{10} & \cdots & \mathbf{0} & \mathbf{B}_{10} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_{n0} & \mathbf{B}_{n0} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{A}_{01} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{B}_{01} & \cdots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{11} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{B}_{11} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_{n1} & \mathbf{0} & \mathbf{B}_{n1} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{A}_{0m} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{B}_{0m} \\ \mathbf{0} & \mathbf{A}_{1m} & \cdots & \mathbf{0} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{B}_{1m} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{A}_{nm} & \mathbf{0} & \mathbf{0} & \cdots & \mathbf{B}_{nm} \end{bmatrix} \tag{2.34}$$

In this subsection, we briefly talked about the "front end" and the "back end" of the visual SLAM. We conduct a detail Jacobian derivation for the pose estimation and implement the tracking process as an example under the G2O framework. We need to use the the pose estimation from the SLAM to fuse the 3D points.

2.3 Visual Inertial Odometry

In the previous section, we discussed pure visual odometry. However, pure visual odometry is not robust to rapid state change like fast rotation. To overcome this shortcoming, we can use the Inertial Measurement Unit (IMU) to acquire the high-frequency inertial measurements such as the acceleration and the angular rate. We can couple the high-frequency IMU and visual measurements to obtain robust pose and map estimation. However, the IMU itself is noisy and has a bias that

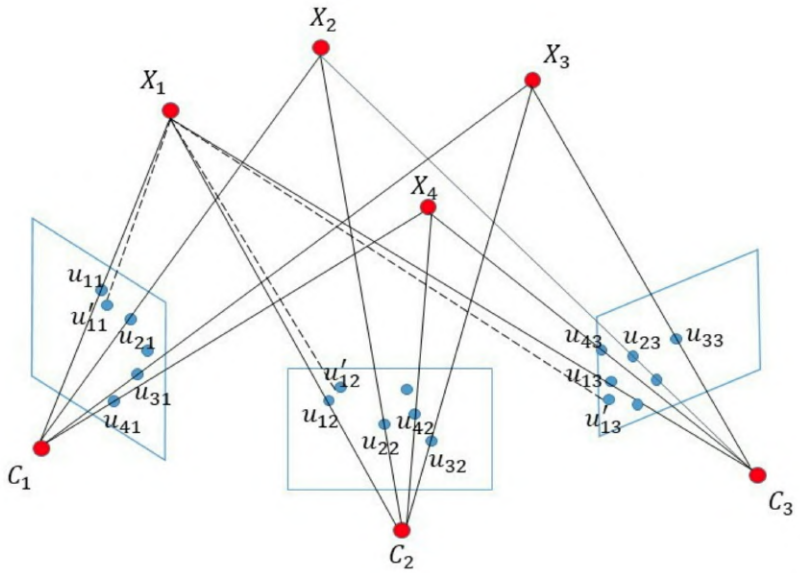


Figure 2.6: In this figure, we have four world landmarks and three frames. Each frame can see all landmarks, which is not true in reality. But it often happens we can see a landmark in several frames. Thus, the landmark can be constraints to all frames and all the frames can also be constraints to all the landmarks. Thus we have the $\mathbb{A}$ and $\mathbb{B}$ matrix in the Eq. (2.32). The bundle adjustment tries to optimize the frames pose as well as the landmarks under the LM framework.

varies slowly. We model the IMU measurement as the following [43]

$$
\begin{aligned}
{}_{\mathrm{B}}\widetilde{\boldsymbol{\omega}}_{\mathrm{WB}}(t) &= {}_{\mathrm{B}}\,\boldsymbol{\omega}_{\mathrm{WB}}(t) + \mathbf{b}^{g}(t) + \boldsymbol{\eta}^{g}(t) \\
{}_{\mathrm{B}}\widetilde{\mathbf{a}} &= \mathbf{R}^{T}_{\mathrm{WB}}(t)({}_{\mathrm{W}}\mathbf{a}(t) - {}_{\mathrm{W}}\,\mathbf{g}) + \mathbf{b}^{a}(t) + \boldsymbol{\eta}^{a}(t)
\end{aligned}
\tag{2.35}
$$

The W represents the world frame, and the B is the body frame, i.e., the IMU frame. In Fig. 2.7. We have one more C stands for the camera frame. $\mathrm{T_{BC}}$ is the transformation matrix between the camera and the body frame. We can use it to map a point in the frame C to B. $\mathrm{T_{WB}}$ is the transformation matrix between the body and the world frame. We typically try to fix the relative position between B and C, i.e., the $\mathrm{T_{BC}}$ is a constant matrix. Obtaining a high resolution $\mathrm{T_{BC}}$ is essential to visual-inertial calibration. This process is called extrinsic calibration, and researchers have developed tools to do extrinsic calibration [90], [44].

In Eq. (2.35), the prefix means the measurement is expressed in the prefix frame. ${}_{\mathrm{B}}\boldsymbol{\omega}_{\mathrm{WB}}$

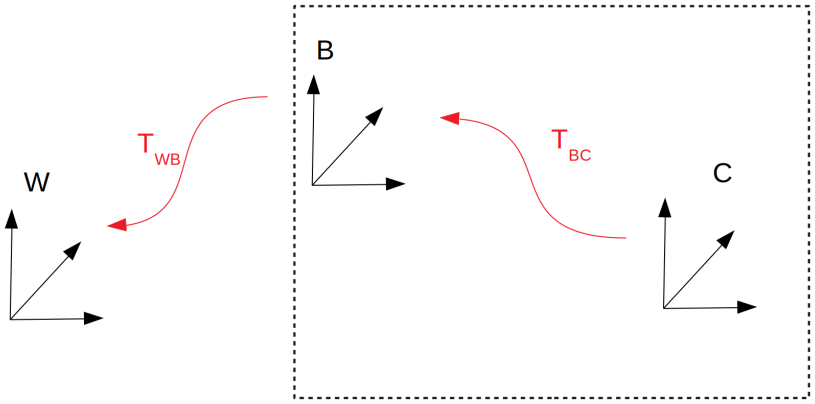


Figure 2.7: World (W), camera (C) and IMU (B) frames. We fix the relative position and orientation between the B and C (in the dashed line). We estimate the transformation between W and B in the visual-inertial odometry.

denotes the "true" angular velocity without noise or bias in the body frame relative to world frame (WB). It is expressed in the body frame (B). ${}_{\mathrm{B}}\boldsymbol{\omega}_{\mathrm{WB}}$ is the noisy and bias measurement, i.e., what the IMU outputs. ${}_{\mathrm{W}}\mathbf{g}$ is the gravity vector in the world frame. $\mathbf{R}^{T}_{\mathrm{WB}}(t)$ is the rotation matrix from body to world. $\mathbf{b}^{g}(t)$ is the bias for gyroscope measurement, $\mathbf{b}^{a}(t)$ is the bias for acceleration measurement. $\boldsymbol{\eta}$ is the white noise. ${}_{\mathrm{B}}\widetilde{\boldsymbol{\omega}}_{\mathrm{WB}}(t) \in \mathbb{R}^3$, same as the other measurements, noise and

bias. For the IMU measurement at time t, we obtain the next sample time $t+\Delta t$ body frame state by (we omit the $\wedge$ operation in the exp for simplification)

$$\begin{aligned}
&\mathbf{R}_{\mathrm{WB}}(t+\Delta t) = \mathbf{R}_{\mathrm{WB}}(t)\exp(\int_{\mathrm{t}}^{\mathrm{t}+\Delta\mathrm{t}} {}_{\mathrm{B}}\boldsymbol{\omega}_{\mathrm{WB}}(\mathrm{t})(\tau)\mathrm{d}\tau) \\
&{}_{\mathrm{W}}\mathbf{v}(t+\Delta t) =_{\mathrm{W}} \mathbf{v}(t) + \int_t^{t+\Delta t} {}_{\mathrm{W}}\mathbf{a}(\tau)d\tau \\
&{}_{\mathrm{W}}\mathbf{p}(t+\Delta t) =_{\mathrm{W}} \mathbf{p}(t) + \int_t^{t+\Delta t} {}_{\mathrm{W}}\mathbf{v}(\tau)d\tau + \iint_t^{t+\Delta t} {}_{\mathrm{W}}\mathbf{a}(\tau)d\tau
\end{aligned} \tag{2.36}$$

In the following equation, we drop the frame subscripts for simplification. Then the above equation is

$$\begin{aligned}
&\mathbf{R}(t+\Delta t) = \mathbf{R}\exp(\int_t^{t+\Delta t} \boldsymbol{\omega}(t)(\tau)d\tau) \\
&\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \int_t^{t+\Delta t} \mathbf{a}(\tau)d\tau \\
&\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \int_t^{t+\Delta t} \mathbf{v}(\tau)d\tau + \iint_t^{t+\Delta t} \mathbf{a}(\tau)d\tau
\end{aligned} \tag{2.37}$$

We assume the acceleration and angular velocity are constants between the two samples since we have high-frequency IMU. Then we have

$$\begin{aligned}
&\mathbf{R}(t+\Delta t) = \mathbf{R}\exp(\boldsymbol{\omega}(t)\Delta t) \\
&\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}\Delta t \\
&\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}\Delta t + \frac{1}{2}\mathbf{a}\Delta t^2
\end{aligned} \tag{2.38}$$

Substitude Eq. (2.35) into the above equation we have

$$\begin{aligned}
&\mathbf{R}(t+\Delta t) = \mathbf{R}\exp((\widetilde{\boldsymbol{\omega}}(t) - \mathbf{b}^g(t) - \boldsymbol{\eta}^{gd}(t))\Delta t) \\
&\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{1}{2}\mathbf{g}\Delta t + \mathbf{R}(t)(\widetilde{\mathbf{a}}(t) - \mathbf{b}^a(t) - \boldsymbol{\eta}^{ad}(t))\Delta t \\
&\mathbf{p}(t+\Delta t) = \mathbf{p}(t) + \mathbf{v}\Delta t + \frac{1}{2}\mathbf{g}\Delta t^2 + \frac{1}{2}\mathbf{R}(t)((\widetilde{\mathbf{a}}(t) - \mathbf{b}^a(t) - \boldsymbol{\eta}^{ad}(t))\Delta t^2
\end{aligned} \tag{2.39}$$

where the $\boldsymbol{\eta}^{ad}(t)$ is the addtive white noise in discrete time. The covariance is amplified by the sample time. For example, if the noise in continuous time is zero mean Gaussian, i.e., $\boldsymbol{\eta}^g(t) \in \mathcal{N}(\mathbf{0}, \mathbf{M})$, then $\boldsymbol{\eta}^{gd}(t) \in \mathcal{N}(\mathbf{0}, \frac{\mathbf{M}}{\Delta t})$ [27], [3].

We usually perform visual-inertial bundle adjustment between two keyframes. There are typically thousands of IMU measurements between two keyframes. Assume we can synchronize the

IMU and the camera. To integrate the measurement of the IMU from keyframe i to keyframe j, we need the following equation based on Eq. (2.39).

$$\begin{aligned}
\mathbf{R}_j &= \mathbf{R}_i \prod_{k=i}^{j-1} \exp((\widetilde{\boldsymbol{\omega}}_k - \mathbf{b}_k^g - \boldsymbol{\eta}_k^{gd})\Delta t) \\
\mathbf{v}_j &= \mathbf{v}_i + \frac{1}{2}\mathbf{g}\Delta t_{ij} + \sum_{k=i}^{j-1} \mathbf{R}_k(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t \\
\mathbf{p}_j &= \mathbf{p}_i + \sum_{k=i}^{j-1}[\mathbf{v}_k\Delta t + \frac{1}{2}\mathbf{g}\Delta t^2 + \frac{1}{2}\mathbf{R}_k((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t^2]
\end{aligned} \tag{2.40}$$

where the k is IMU sample time, $k+1$ is the next IMU sample time. From $k=i$ to $k=j$ we integrate all the IMU measurements between two keyframes. However, the above equation is not practical in a real-world application. Recall the normal equation in Eq. (2.29). The δx is the state updates after one iteration. Assume the state contains the $\mathbf{R}, \mathbf{v}, \mathbf{p}$ of two keyframes. Then for each iteration, we need to integrate the above equation with thousands of measurements at once, and we typically need more than one iteration. As the integration is expensive, we want to perform it as little as possible. Then we define the following new variables.

$$\begin{aligned}
\Delta\mathbf{R}_{ij} &\doteq \mathbf{R}_i^T\mathbf{R}_j = \prod_{k=i}^{j-1} \exp((\widetilde{\boldsymbol{\omega}}_k - \mathbf{b}_k^g - \boldsymbol{\eta}_k^{gd})\Delta t) \\
\Delta\mathbf{v}_{ij} &\doteq \mathbf{R}_i^T(\mathbf{v}_j - \mathbf{v}_i - \frac{1}{2}\mathbf{g}\Delta t_{ij}) = \sum_{k=i}^{j-1} \Delta\mathbf{R}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t \\
\Delta\mathbf{p}_{ij} &\doteq \mathbf{R}_i^T(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i\Delta t_{ij} - \frac{1}{2}\mathbf{g}\Delta t_{ij}) = \sum_{k=i}^{j-1}[\Delta\mathbf{v}_{ik}\Delta t + \frac{1}{2}\Delta\mathbf{R}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t^2]
\end{aligned} \tag{2.41}$$

where the $\Delta\mathbf{R}_{ik}$ and $\Delta\mathbf{v}_{ik}$ are

$$\begin{aligned}
\Delta\mathbf{R}_{ik} &= \prod_{k=i}^{m-1} \exp((\widetilde{\boldsymbol{\omega}}_k - \mathbf{b}_k^g - \boldsymbol{\eta}_k^{gd})\Delta t) \\
\Delta\mathbf{v}_{ik} &= \sum_{k=i}^{m-1} \Delta\mathbf{R}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t
\end{aligned} \tag{2.42}$$

$m \in (i, j)$. Bring Eq. (2.42) into Eq. (2.41). We notice that all the equations on the right side of Eq. (2.41) are independent from $\mathbf{R}_i, \mathbf{R}_j, \mathbf{v}_i, \mathbf{v}_j, \mathbf{p}_i, \mathbf{p}_j$. We can decide the $\Delta\mathbf{R}_{ij}, \Delta\mathbf{v}_{ij}, \Delta\mathbf{p}_{ij}$ by IMU measurements $\widetilde{\boldsymbol{\omega}}, \widetilde{\mathbf{a}}$ and the bias, noise. Thus, if we integrate the Eq (2.41) in advance, we can treat them as "*one*" no matter how the normal equation updates the sensor state. If we want

to calculate $\mathbf{R}_j$ we simply need to perform $\mathbf{R}_i\Delta\mathbf{R}_{ij}$, for example. Eq. (2.40) and the Eq. (2.41) are essentially the same. We just extract parts that are independent from the states in the normal equation to avoid re-integration. This approach is well-known as *preintegration* shown in Fig. 2.8.

However, the current preintegration items contain bias. The bias changes slowly, so we

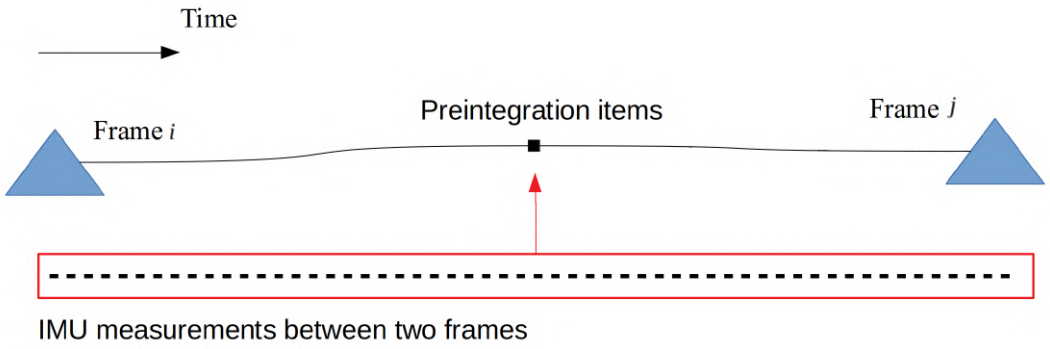


Figure 2.8: We use IMU measurements as the extra constraints between frames. To avoid re-propagation of the multiple IMU items, we unify them as "preintegration items". They are the $\Delta\mathbf{R}_{ij}, \Delta\mathbf{v}_{ij}, \Delta\mathbf{p}_{ij}$ in Eq. (2.41).

typically need to include them in the optimization states. Thus, the preintegration items still include optimization states in the normal equation. There are various solutions. If the bias updates are small, we can use the preintegration's first-order expansion to update the preintegration items. However, if they are too large, we may have to re-propagation [85], [43], [84]. In the following sections, we first assume the bias is constant between two keyframes, so we don't have to update them in the normal equation. Then, we discuss the modifications we need to conduct considering the bias update.

Besides the bias, the white noise items $\boldsymbol{\eta}$ also bring troubles because we don't know their "true" values. To solve this problem, we hope the preintegration items can exclude the noise items.

To do this, we conduct the following derivations for the preintegration terms. Firstly, the $\Delta\mathbf{R}_{ij}$.

$$
\begin{aligned}
\Delta\mathbf{R}_{ij} &= \prod_{k=i}^{j-1} \exp((\widetilde{\boldsymbol{\omega}}_k - \mathbf{b}_k^g - \boldsymbol{\eta}_k^{gd})\Delta t) \\
&\overset{(1)}{\approx} \prod_{k=i}^{j-1} [\exp((\tilde{\boldsymbol{\omega}}_k - \mathbf{b}_i^g)\Delta t)\exp(-J_r^k \boldsymbol{\eta}_k^{gd}\Delta t)] \\
&\overset{(2)}{=} \exp((\tilde{\boldsymbol{\omega}}_i - \mathbf{b}_i^g)\Delta t)\exp(-J_r^i \boldsymbol{\eta}_i^{gd}\Delta t) * \exp((\tilde{\boldsymbol{\omega}}_{i+1} - \mathbf{b}_{i+1}^g)\Delta t)\exp(-J_r^{i+1}\boldsymbol{\eta}_{i+1}^{gd}\Delta t) \\
&* \exp((\tilde{\boldsymbol{\omega}}_{i+2} - \mathbf{b}_{i+2}^g)\Delta t)\exp(-J_r^{i+2}\boldsymbol{\eta}_{i+2}^{gd}\Delta t)... \\
&\overset{(3)}{=} \exp(A_i)\exp(B_i) * \exp(A_{i+1})\exp(B_{i+1}) * \exp(A_{i+2})\exp(B_{i+2})... \\
&\overset{(4)}{=} \mathbf{R}_{A_i}\exp(B_i) * \mathbf{R}_{A_{i+1}}\exp(B_{i+1}) * \mathbf{R}_{A_{i+2}}\exp(B_{i+2}) * \mathbf{R}_{A_{i+3}}\exp(B_{i+3})... \\
&\overset{(5)}{=} \mathbf{R}_{A_i}\mathbf{R}_{A_{i+1}}\exp(\mathbf{R}_{A_{i+1}}^T B_i)\exp(B_{i+1}) * \mathbf{R}_{A_{i+2}}\exp(B_{i+2}) * \mathbf{R}_{A_{i+3}}\exp(B_{i+3})... \\
&\overset{(6)}{=} \mathbf{R}_{A_i}\mathbf{R}_{A_{i+1}}\exp(\mathbf{R}_{A_{i+1}}^T B_i)\mathbf{R}_{A_{i+2}}\exp(\mathbf{R}_{A_{i+2}}^T B_{i+1})\exp(B_{i+2}) * \mathbf{R}_{A_{i+3}}\exp(B_{i+3})... \\
&\overset{(7)}{=} \mathbf{R}_{A_i}\mathbf{R}_{A_{i+1}}\mathbf{R}_{A_{i+2}}\exp(\mathbf{R}_{A_{i+2}}^T\mathbf{R}_{A_{i+1}}^T B_i)\exp(\mathbf{R}_{A_{i+2}}^T B_{i+1})\exp(B_{i+2}) * \mathbf{R}_{A_{i+3}}\exp(B_{i+3})... \\
&... \\
&\overset{(8)}{=} \mathbf{R}_{A_i}\mathbf{R}_{A_{i+1}}...\mathbf{R}_{A_{j-1}} * \exp(\mathbf{R}_{A_{j-1}}^T...\mathbf{R}_{A_{i+1}}^T B_i) * \exp(\mathbf{R}_{A_{j-1}}^T...\mathbf{R}_{A_{i+2}}^T B_{i+1})... \\
&\overset{(9)}{\doteq} \Delta\widetilde{\mathbf{R}}_{ij}\prod_{k=i}^{j-1}\exp(\Delta\widetilde{\mathbf{R}}_{k+1j}B_i) \\
&\overset{(10)}{\doteq} \Delta\widetilde{\mathbf{R}}_{ij}\exp(-\delta\boldsymbol{\theta}_{ij})
\end{aligned}
\tag{2.43}
$$

In the subequation (1), we utilize the third property in Eq. (2.22). We extract the noise term to the right exponential map. In the subequation (2) we expand this equation with three terms, from $i, i+1, i+2$ to the end. In the subequation (3) we redefine $A_i \doteq (\tilde{\boldsymbol{\omega}}_i - \mathbf{b}_i^g)\Delta t$ and $B_i \doteq -J_r^{i+1}\boldsymbol{\eta}_{i+1}^{gd}\Delta t$ for simplification. From the subequation (3) to (4), we re-define $\mathbf{R}_{A_i} = \exp(A_i)$ but we don't re-define the $\exp(B_i)$ to make the following prove more clear. According to the fourth property $\exp(\boldsymbol{\theta}^\wedge)\mathbf{R} = \mathbf{R}\exp((\mathbf{R}^T\boldsymbol{\theta})^\wedge)$ in Eq. (2.22), we can move the $\mathbf{R}_{A_{i+1}}$ to the left side of the $\exp(B_i)$ then we have the subequation (5). Continuously, we apply the above property and move $\mathbf{R}_{A_{i+2}}$ twice and we have the subequation (6) and (7). By doing the above procedure repeatly, we finally move all $\mathbf{R}_A$ to the left side and leave the exponential term to the right side, which

is the subequation (8). After that, we re-define $\Delta\widetilde{\mathbf{R}}_{ij} \doteq \mathbf{R}_{A_i}...\mathbf{R}_{A_{j-1}}$, then we can apply the same definition $\Delta\widetilde{\mathbf{R}}_{k+1j} \doteq \mathbf{R}^T_{A_{j-1}}...\mathbf{R}^T_{A_{i+1}}$. Then we have the subequation (9). Finally, we treat $\Delta\widetilde{\mathbf{R}}_{k+1j}B_i$ as a unified noise term $-\boldsymbol{\theta}_{ij}$ (there is a - sign in the B_i), we have the subequation (10).

We can calculate $\Delta\widetilde{\mathbf{R}}_{ij} = \prod_{k=i}^{j-1}\exp((\widetilde{\boldsymbol{\omega}}_k - \mathbf{b}^g_i)\Delta t)$ with little efforts since it only contains the bias and IMU measurements. Thus, we define the $\Delta\widetilde{\mathbf{R}}_{ij}$ as the preintegrated rotation measurements and its corresponding noise term $\delta\boldsymbol{\theta}_{ij}$.

Bring the result of Eq. (2.43) into the velocity term of Eq. (2.41).

$$
\begin{aligned}
\Delta\mathbf{v}_{ij} &= \sum_{k=i}^{j-1}\Delta\mathbf{R}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k - \boldsymbol{\eta}^{ad}_k)\Delta t \\
&\overset{(1)}{\approx} \sum_{k=i}^{j-1}\Delta\widetilde{\mathbf{R}}_{ik}\exp(-\delta\boldsymbol{\theta}^{\wedge}_{ik})(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k - \boldsymbol{\eta}^{ad}_k)\Delta t \\
&\overset{(2)}{\approx} \sum_{k=i}^{j-1}\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\boldsymbol{\theta}^{\wedge}_{ik})(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k - \boldsymbol{\eta}^{ad}_k)\Delta t \\
&\overset{(3)}{=} \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\boldsymbol{\theta}^{\wedge}_{ik})(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\boldsymbol{\theta}^{\wedge}_{ik})\boldsymbol{\eta}^{ad}_k\Delta t] \\
&\overset{(4)}{\approx} \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\boldsymbol{\theta}^{\wedge}_{ik})(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}^{ad}_k\Delta t] \\
&\overset{(5)}{=} \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\delta\boldsymbol{\theta}^{\wedge}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}^{ad}_k\Delta t] \\
&\overset{(6)}{=} \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t + \Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)^{\wedge}\delta\boldsymbol{\theta}_{ik}\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}^{ad}_k\Delta t] \\
&\overset{(7)}{=} \sum_{k=i}^{j-1}\Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t + \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)^{\wedge}\delta\boldsymbol{\theta}_{ik}\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}^{ad}_k\Delta t] \\
&\overset{(8)}{\doteq} \Delta\widetilde{\mathbf{v}}_{ij} - \delta\mathbf{v}_{ij}
\end{aligned}
\tag{2.44}
$$

From the subequation (1) to (2), we use the second property in Eq. (2.22). From the subequation (3) to (4), we omit high order noise term that includes the production of $\delta\boldsymbol{\theta}\boldsymbol{\eta}$. Finally we define the $\widetilde{\mathbf{v}}_{ij} \doteq \sum_{k=i}^{j-1}\Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}^a_k)\Delta t$ as the preintegrated velocity measurement. The rest $\delta\mathbf{v}_{id}$ is the velocity noise term.

Substitude the result of Eq. (2.44) and (2.43) into Eq. (2.41) position term, we can have the

noise-free preintegrated position measurement term.

$$
\begin{aligned}
\Delta\mathbf{p}_{ij} &= \sum_{k=i}^{j-1}[\Delta\mathbf{v}_{ik}\Delta t + \frac{1}{2}\Delta\mathbf{R}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t^2] \\
&= \sum_{k=i}^{j-1}[(\Delta\widetilde{\mathbf{v}}_{ik} - \delta\mathbf{v}_{ik})\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}\exp(-\delta\boldsymbol{\theta}_{ik}^{\wedge})((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t^2] \\
&= \sum_{k=i}^{j-1}[(\Delta\widetilde{\mathbf{v}}_{ik} - \delta\mathbf{v}_{ik})\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{I} - \delta\boldsymbol{\theta}_{ik}^{\wedge})((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a - \boldsymbol{\eta}_k^{ad})\Delta t^2] \\
&= \sum_{k=i}^{j-1}[\Delta\widetilde{\mathbf{v}}_{ik}\Delta t - \delta\mathbf{v}_{ik}\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2 - \Delta\widetilde{\mathbf{R}}_{ik}\delta\boldsymbol{\theta}_{ik}^{\wedge}(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2 - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}_k^{ad}\Delta t^2] \\
&= \sum_{k=i}^{j-1}[(\Delta\widetilde{\mathbf{v}}_{ik}\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2) - \delta\mathbf{v}_{ik}\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}\delta\boldsymbol{\theta}_{ik}^{\wedge}(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2 - \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}_k^{ad}\Delta t^2] \\
&= \sum_{k=i}^{j-1}(\Delta\widetilde{\mathbf{v}}_{ik}\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2) \\
&- \sum_{k=i}^{j-1}(\delta\mathbf{v}_{ik}\Delta t - \Delta\widetilde{\mathbf{R}}_{ik}(\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)^{\wedge}\delta\boldsymbol{\theta}_{ik}\Delta t^2 + \Delta\widetilde{\mathbf{R}}_{ik}\boldsymbol{\eta}_k^{ad}\Delta t^2)] \\
&\doteq \Delta\widetilde{\mathbf{p}}_{ij} - \delta\mathbf{p}_{ij}
\end{aligned}
\tag{2.45}
$$

We derive the above equation using the same properties in Eq. (2.44). We define the $\Delta\widetilde{\mathbf{p}}_{ij} = \sum_{k=i}^{j-1}(\Delta\widetilde{\mathbf{v}}_{ik}\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}((\widetilde{\mathbf{a}}_k - \mathbf{b}_k^a)\Delta t^2)$ as the preintegrated position measurement.

So far, we have successfully conducted the rotation, velocity, and position preintegrated terms. Those preintegrated terms only contain the bias and the IMU measurements. Then we can have our IMU preintegration model with noise. Substitute the results of Eq. (2.43), (2.44), (2.43) into Eq. (2.41).

$$
\begin{aligned}
\delta\boldsymbol{\theta}_{ij} &= \log(\Delta\widetilde{\mathbf{R}}_{ij}\mathbf{R}_j^T\mathbf{R}_i)^{\vee} \\
\delta\mathbf{v}_{ij} &= \Delta\widetilde{\mathbf{v}}_{ij} - (\mathbf{R}_i^T(\mathbf{v}_j - \mathbf{v}_i - \frac{1}{2}\mathbf{g}\Delta t_{ij})) \\
\delta\mathbf{p}_{ij} &= \Delta\widetilde{\mathbf{p}}_{ij} - (\mathbf{R}_i^T(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i\Delta t_{ij} - \frac{1}{2}\mathbf{g}\Delta t_{ij}))
\end{aligned}
\tag{2.46}
$$

On the left side are the error terms. Comparing the pure visual odometry error term, in which we only use the pixel intensity/position difference (Eq. (2.9)) as the error, the IMU preintegrated model error is more complicated. However, the idea behind the scene is straightforward. We can calculate the preintegrated terms simply by the IMU with measurements and bias propagation. We

can have the $\mathbf{R}_j, \mathbf{v}_j, \mathbf{p}_j$ using the constant velocity model. They are the initial guess of the states. Then we try to find the best sensor transformation to minimize the error term, which brings us back to the GN/LM framework.

Now we can consider the bias term updates. We know the preintegrated measurements consider the bias and the IMU measurements. During the LM iteration, we update the bias $\bar{\mathbf{b}}_i^g$ with a small value $\delta\mathbf{b}$ and get $\mathbf{b}_i^g$. If we want to avoid repropagation, we use the first order Jacobian of the $\Delta\widetilde{\mathbf{R}}_{ij}$ w.r.t the bias to update the $\Delta\widetilde{\mathbf{R}}_{ij}$. Same as the velocity and the position.

$$\begin{aligned}
\Delta\widetilde{\mathbf{R}}_{ij}(\mathbf{b}_i^g) &\approx \Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}^g) \\
\Delta\widetilde{\mathbf{v}}_{ij}(\mathbf{b}_i^g, \mathbf{b}_i^a) &\approx \Delta\widetilde{\mathbf{v}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a \\
\Delta\widetilde{\mathbf{p}}_{ij}(\mathbf{b}_i^g, \mathbf{b}_i^a) &\approx \Delta\widetilde{\mathbf{p}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a
\end{aligned} \tag{2.47}$$

According to the definition of the preintegrated term, we can easily have the partial derivatives

$$\begin{aligned}
\Delta\widetilde{\mathbf{R}}_{ij}(\mathbf{b}_i^g) &\overset{(1)}{=} \prod_{k=i}^{j-1}\exp((\widetilde{\boldsymbol{\omega}}_k - \bar{\mathbf{b}}_i^g - \delta\mathbf{b}_i^g)\Delta t \\
&\overset{(2)}{\approx} \prod_{k=i}^{j-1}\exp((\widetilde{\boldsymbol{\omega}}_k - \bar{\mathbf{b}}_i^g)\Delta t)\exp(-J_r^k\delta\mathbf{b}_i^g\Delta t) \\
&\overset{(3)}{=} \Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\prod_{k=i}^{j-1}\exp([-\Delta\widetilde{\mathbf{R}}_{k+1j}(\bar{\mathbf{b}}_i)^T J_r^k\Delta t]\delta\mathbf{b}_i^g) \\
&\overset{(4)}{\approx} \Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp([\sum_{k=i}^{j-1}(-\Delta\widetilde{\mathbf{R}}_{k+1j}(\bar{\mathbf{b}}_i)^T J_r^k\Delta t)]\delta\mathbf{b}_i^g) \\
&\overset{(5)}{=} \Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)
\end{aligned} \tag{2.48}$$

This equation uses several previous non-trivial proofs and approximations. From the subequation (1) to (3), we apply the exactly same procedure as Eq. (2.43). From (3) to (4), we use the approximation from Eq. (2.25). As we can see from the above proof, the partial derivative w.r.t $\mathbf{b}^g$ is

$$\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g} = \sum_{k=i}^{j-1} -\Delta\widetilde{\mathbf{R}}_{k+1j}(\bar{\mathbf{b}}_i)^T J_r^k\Delta t \tag{2.49}$$

. For the velocity term, we have

$$
\begin{aligned}
\Delta\widetilde{\mathbf{v}}_{ij}(\mathbf{b}_i^g, \mathbf{b}_i^a) &\overset{(1)}{=} \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \mathbf{b}_i^a)\Delta t \\
&\overset{(2)}{\approx} \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a - \delta\mathbf{b}_i^a)\Delta t \\
&\overset{(3)}{\approx} \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\mathbf{I} + \frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a - \delta\mathbf{b}_i^a)\Delta t \\
&\overset{(4)}{\approx} \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)\Delta t + \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)\Delta t\delta\mathbf{b}_i^g - \\
&\quad \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\Delta t\delta\bar{\mathbf{b}}_i^a \\
&\doteq \widetilde{\mathbf{v}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a
\end{aligned}
\tag{2.50}
$$

From the sub-equation (2) to (3), we use the second property of the Eq. (2.22). We also remove the high order term with the multiplication $\delta\mathbf{b}_i^g\delta\mathbf{b}_i^a$ from the sub equation (3) to (4). The partial derivative of the velocity w.r.t the bias is

$$
\begin{aligned}
\frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^g} &= \sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)\Delta t\delta\mathbf{b}_i^g \\
&= -\sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)^\wedge\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\Delta t\delta\mathbf{b}_i^g \\
\frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^a} &= -\sum_{k=i}^{j-1} \Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\Delta t
\end{aligned}
\tag{2.51}
$$

For the position term, we have

$$
\begin{aligned}
\Delta\widetilde{\mathbf{p}}_{ij}(\mathbf{b}_i^g, \mathbf{b}_i^a) &= \sum_{k=i}^{j-1}(\Delta\widetilde{\mathbf{v}}_{ik}\Delta t + \frac{1}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \mathbf{b}_i^a)\Delta t^2) \\
&= \sum_{k=i}^{j-1}(\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \mathbf{b}_i^a)\Delta t^2) \\
&= \sum_{k=i}^{j-1}(\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a - \delta\mathbf{b}_i^a)\Delta t^2) \\
&\approx \sum_{k=i}^{j-1}(\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a - \delta\mathbf{b}_i^a)\Delta t^2) \\
&\approx \sum_{k=i}^{j-1}(\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\mathbf{I} + \frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a - \delta\mathbf{b}_i^a)\Delta t^2) \\
&= \sum_{k=i}^{j-1}\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)\Delta t^2 - \sum_{k=i}^{j-1}\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)^\wedge\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\Delta t^2\delta\mathbf{b}_i^g \\
&\quad - \sum_{k=i}^{j-1}\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\Delta t^2\delta\mathbf{b}_i^a \\
&\doteq \Delta\widetilde{\mathbf{p}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a
\end{aligned}
\tag{2.52}
$$

The partial derivative of the position w.r.t the bias is

$$
\begin{aligned}
\frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^g} &= -\sum_{k=i}^{j-1}\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)(\widetilde{\mathbf{a}}_k - \bar{\mathbf{b}}_i^a)^\wedge\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\Delta t^2 \\
\frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^a} &= -\sum_{k=i}^{j-1}\frac{3}{2}\Delta\widetilde{\mathbf{R}}_{ik}(\bar{\mathbf{b}}_i^g)\Delta t^2
\end{aligned}
\tag{2.53}
$$

So far, we have all the preintegrated terms and their noise considering the bias update. Bring the preintegrated terms back into the definition Eq. (2.41). The differences between the preintegrated terms and the initial guess are our error terms, i.e. the noise terms of the results in Eq. (2.43), (2.44), (2.45) $-\delta\boldsymbol{\theta}_{ij}, -\delta\mathbf{v}_{ij}, -\delta\mathbf{p}_{ij}$. We move all the noise terms into the equation left side and define them as the residuals. $-\delta\boldsymbol{\theta}_{ij} \doteq \mathbf{r}_{\Delta\mathbf{R}_{ij}}, -\delta\mathbf{v}_{ij} \doteq \mathbf{r}_{\Delta\mathbf{v}_{ij}}, -\delta\mathbf{p}_{ij} \doteq \mathbf{r}_{\Delta\mathbf{p}_{ij}}$

$$
\begin{aligned}
&\mathbf{R}_{ij} \doteq \mathbf{R}_i^T\mathbf{R}_j \approx \Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)\exp(-\delta\boldsymbol{\theta}_{ij}) \\
&\rightarrow \\
&-\delta\boldsymbol{\theta}_{ij} = \log(\Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g))^T\mathbf{R}_i^T\mathbf{R}_j
\end{aligned}
\tag{2.54}
$$

Use the similar resoning, we have all the residuals terms for LM algorithm

$$\begin{aligned}
\mathbf{r}_{\Delta\mathbf{R}_{ij}} &\doteq \log[\Delta\widetilde{\mathbf{R}}_{ij}(\bar{\mathbf{b}}_i^g)\exp(\frac{\partial\Delta\bar{\mathbf{R}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g)]^T\mathbf{R}_i^T\mathbf{R}_j \\
\mathbf{r}_{\Delta\mathbf{v}_{ij}} &\doteq \mathbf{R}_i^T(\mathbf{v}_j - \mathbf{v}_i - \frac{1}{2}\mathbf{g}\Delta t_{ij}) - [\Delta\widetilde{\mathbf{v}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{v}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a] \\
\mathbf{r}_{\Delta\mathbf{p}_{ij}} &\doteq \mathbf{R}_i^T(\mathbf{p}_j - \mathbf{p}_i - \mathbf{v}_i\Delta t_{ij} - \frac{1}{2}\mathbf{g}\Delta t_{ij}) - [\Delta\widetilde{\mathbf{p}}_{ij}(\bar{\mathbf{b}}_i^g, \bar{\mathbf{b}}_i^a) + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^g}\delta\mathbf{b}_i^g + \frac{\partial\Delta\bar{\mathbf{p}}_{ij}}{\partial\mathbf{b}^a}\delta\mathbf{b}_i^a]
\end{aligned} \tag{2.55}$$

We can compute the initial $\mathbf{R}_i, \mathbf{R}_j, \mathbf{v}_i, \mathbf{v}_i, \mathbf{p}_j$ using pure visual odometry; we can calculate the preintegrated terms only with IMU measurements. Thus, it is straightforward to obtain the error terms with the above formula. To throw them into a GN/LM optimization framework, we also need the Jacobian matrix of the error function w.r.t the tracking states. i.e.,

$$\begin{aligned}
&[\frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\boldsymbol{\theta}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\boldsymbol{\theta}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\mathbf{p}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\mathbf{p}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\mathbf{v}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\mathbf{v}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\mathbf{b}_i^a}, \frac{\partial\mathbf{r}_{\Delta\mathbf{R}_{ij}}}{\partial\delta\mathbf{b}_i^g}] \\
&[\frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\boldsymbol{\theta}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\boldsymbol{\theta}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\mathbf{p}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\mathbf{p}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\mathbf{v}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\mathbf{v}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\mathbf{b}_i^a}, \frac{\partial\mathbf{r}_{\Delta\mathbf{v}_{ij}}}{\partial\delta\mathbf{b}_i^g}] \\
&[\frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\boldsymbol{\theta}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\boldsymbol{\theta}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\mathbf{p}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\mathbf{p}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\mathbf{v}_i}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\mathbf{v}_j}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\mathbf{b}_i^a}, \frac{\partial\mathbf{r}_{\Delta\mathbf{p}_{ij}}}{\partial\delta\mathbf{b}_i^g}]
\end{aligned} \tag{2.56}$$

Note to implement the visual inertial Bundle Adjustment, we need the above Jacobian w.r.t the sensor states (corresponding to the $\mathbb{A}$ in Eq. (2.33)) along with the Jacobian w.r.t the visual landmarks ($\mathbb{B}$ in Eq. (2.33)). [42] lists all the Jacobian terms and derivation.

This chapter discusses the fundamental knowledge of visual/visual-inertial navigation and derives the preintegrated error function in detail. Besides the equations, we primarily demonstrate the pose estimation problem under the G2O framework and release the example code.

Chapter 3

Image segmentation

In this chapter, we discuss the image segmentation system for detecting the dynamic objects and present a coustomized U-NET for demonstration. As we have shown in Chapter 1, the general object detection returns a bounding box of a specific object, the probability of different objects (class). The input of a CNN is an image (matrix), and the output is a vector containing class ID and its probability, bounding box length, width, and center. Unlike how human beings visualize the world, the computer just saves what it "sees" as a matrix in its memory. It is easy to teach a human to recognize an object, while it is a different story to "teach" a computer to learn an object.

Before 2012, there were two main directions to solve object detection problems. One is the traditional method, people tried to combine the machine learning classifier such as Support Vector Machines [109], and K-Nearest Neighbours with feature detectors such as SIFT and SURF [111, 10] to implement object detection. The other direction is to use the deep learning-based convolutional neural networks (CNN) based technique. In 2012, the ImageNet Large Scale Visual Recognition Competition achieved a historical low error rate from AlexNet [59], which uses a CNN model to dominate the competition. Since then, all the winners have used a deep learning-based approach, and the error rate has dropped to a few percent. Also, since then, the mainstream object detection work has shifted to a deep learning-based system.

A successful convolutional neural network was released years after 1989's first version, the LeNet5 [62]. However, two main obstacles stop deep learning from becoming a mainstream player

until the very last decade.

The first reason is we lack enough training data. The deep learning-based work typically needs a large training dataset to increase precision. ImageNet [32] (since 2010) provides over 1 million labeled images and over 1000 different classes, which is enough for the training process. Previously, datasets such as Pascal VOC [38] only provides 20,000 images and 20 objects.

The second limitation is the hardware. Image processing requires a large amount of parallel computation, which is often done on GPU. Early GPU cannot provide enough memory nor speed for large dataset training. In 2010, for example, the flag GeForce series GPU GTX 480 released by Nvidia has about 1.5 GB memory and can do about 1344.96 GFLOPS (A 1 GFLOPS computer system is capable of performing one billion floating-point operations per second). In 2020, the flag GeForce GPU GTX 3090 has 24 GB memory and can do about 35.68 TFLOPS (1 TFLOPS = 1000 GFLOPS). Before 2010, the GPU is much less powerful. Hardware improvements make it more accessible to use GPU on a deep learning task.

In this work, we'll introduce deep learning-based work for image detection and segmentation. The detection is relatively easier work to segmentation because we don't need to mask every pixel that belongs to an object.

CNN is the base of the modern object detection. Early work such as LeNet5 uses the basic CNN to identify the digit in a 32 by 32 image. However, such a pure CNN network can only detect one object at a time, and if multiple objects overlap, the detection typically fails. In 2013, R-CNN [46] was proposed to identify various objects in an image. R-CNN firstly generates numbers of sub-regions (we call it regions of interests(RoI)) to segment the image. Then it combines similar regions to a larger one. Then CNN extracts features of every region. Finally, we send the feature to the Support Vector Machine (SVM) for classification, and the regressor is used for bounding box location. The process can be seen in figure 3.1. The main drawback of R-CNN is it is slow. It takes more than 10 seconds to classify the image on their GPU. The main reason for this is that the proposed region of an image is 2000, which costs a long time to use CNN to extract features of each region and then apply the features to SVM. Fast-R-CNN [45], based on R-CNN, overcomes

its drawbacks. The improvements are mainly three aspects

- Use one CNN for feature extraction. The R-CNN needs 2000 CNN for 2000 RoI, which is time-consuming.
- Use softmax to take the place of SVM for classification. However, the result using these two classifiers is comparable.
- Combine the training loss. R-CNN train the classification error but use the regressor to decide bounding box location. The Fast-R-CNN combines the error between bounding box estimation and object classification, which is called multi-task loss.

Based on those modifications, the Fast-R-CNN improvements classification precision as well as the speed. Typically, the prediction process is a hundred milliseconds level on a powerful GPU. However, they didn't improve the expensive selective search method for choosing the initial RoI. The Faster-R-CNN [91] is proposed to overcome the problem. Faster-R-CNN uses a region proposal network to generate the potential regions that have objects. Then the rest is a Fast-R-CNN detector for classification. In short, R-CNN, Fast-R-CNN, and Faster-R-CNN share the same basics. They are all region-based networks. Although they have recently improved, their same basics also become the bottleneck of their speed and accuracy.

What we need in this project is image segmentation work that can finish tasks shown in

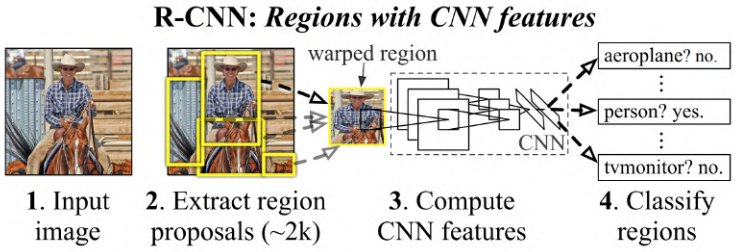


Figure 3.1: R-CNN steps: 1, Input the image. 2 and 3: Selective search Algorithm proposes regions and combines small regions into a large one. 4:Extract feature by CNN. 5: SVM for feature classification

Figure 1.4. Based on Faster-R-CNN, Mask-R-CNN is proposed to do segmentation. Based on the

object detection with a bounding box, adding a mask to the object is fairly easy afterward. On the assumption in each bounding box, there is just one main class. We need to find which pixel-class correspondence. The Mask-R-CNN's drawback is its speed, usually 3 to 5 FPS on a single GPU. We will have a detailed discussion about the network structure since we need to apply an image segmentation network into our system.

Faster RCNN includes two stages, the first stage is to use Region Proposal Networks (RPN) to propose ROI, and then the second stage is to do classification based on the proposed ROI. Figure 3.2 [91] shows its first stage. A convolutional layer is used to generate feature maps for the RPN. This convolutional layer could have various choices. They are the so-called "backbones". The VGG16 [104], for instance.

VGG is the abbreviation of Visual Geometry Group, University of Oxford. The number following VGG is the number of convolutional layers and fully connected layers. The VGG16 has 13 convolutional layers and 3 fully connected layers. An example of VGG is shown in Figure 3.3 [41]. The standard combination convolutional network + ReLU + max-pooling forms one VGG layer. CNN is used to extract features, ReLU is used to add a non-linear property to the network, and max-pooling is used for downsampling (decreasing parameter dimensions to avoid overfitting), translation/rotation invariance. Combining a fully connected network with ReLU is used to map high-level features into vectors (since it normally flattens the input tensor) for classification. The Faster RCNN application uses the convolution part of the VGG initially because it intends to use a fully connected layer after RPN and ROI pooling for final classification. The convolution part of the VGG generates features maps and input to the RPN. In the Pytorch Faster RCNN implementation, the VGG has 13 convolutional layers, 4 ReLu, and 4 Max Pooling. Each Max pooling has kernel size $2x2$, so after each pooling, the image is down-sampled by 4. The output from the VGG in the Faster RCNN for a $M * N$ image is a $(M/16) * (N/16)$ feature map.

ResNet can be another choice of the "backbones". The ResNet also combines multiple convolutional layers. The main difference between ResNet and VGG is it "shorts" the deep layer with shallow layer's output. An example is shown in Figure 3.4 [115], [52]. After the second "weight

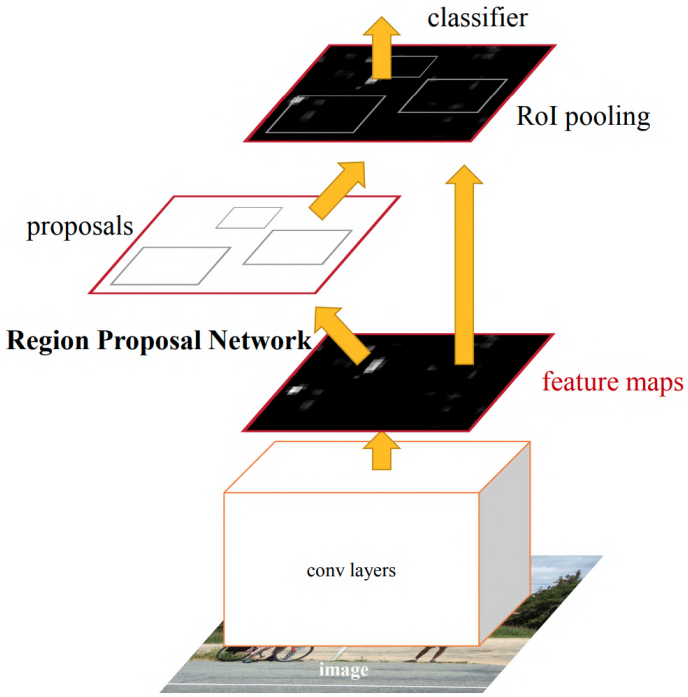


Figure 3.2: Faster RCNN has two main stages. RPN proposes regions of interests (ROI) and then classifier

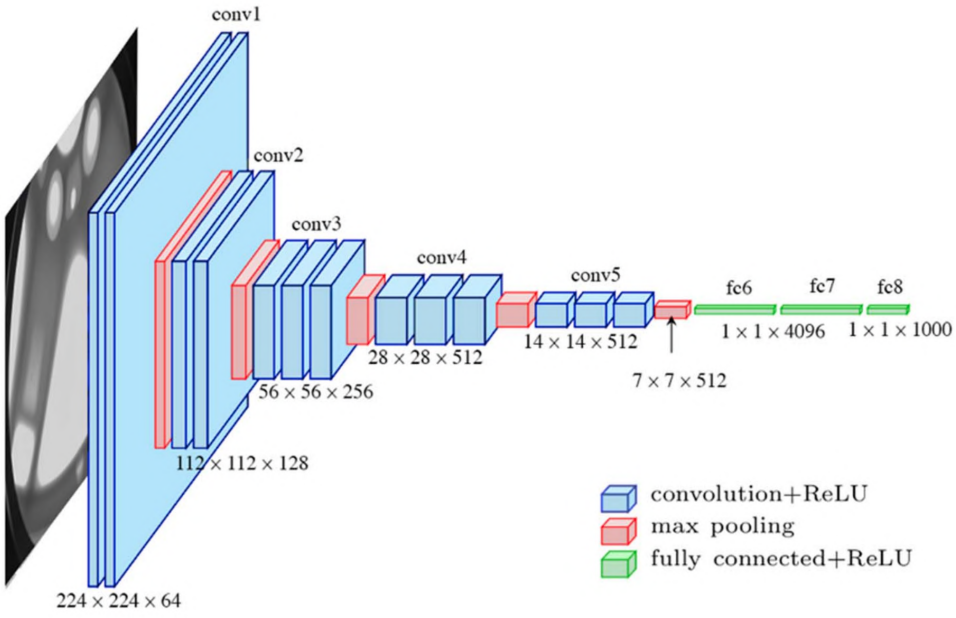


Figure 3.3: VGG network has convolutional layers and fully connected layers.

layer", if we mark the desired mapping as $H(x)$. Then we have $H(x) = F(x) + x$. The residual block tries to learn $F(x) = H(x) - x$, the so-called "residual". This structure is useful when the network has a large number of layers. Previous researchers found more layers don't always help, and it sometimes leads to a worse result. This phenomenon is known as degradation. It is mainly because of the gradient vanishing/exploding problem. The residual block can relieve this problem. A straightforward understanding is that the result won't be worse than the shallow layer since we short the deep and shallow layers. Famous ResNet architechture can be ResNet-34, ResNet-50, ResNet-101, etc. The number after the ResNet stands for the number of the convolutional layers and fully connected layers. Figure 3.6 , [73] shows an example of ResNet-50. If the input is a 224 by 224, 3 channel image, the first convolution block contains a single convolutional network, and it has a size 7 kernel, the second convolution block ("Conv2") has three groups. Each group contains 3 convolution networks with kernel sizes 1,3, and 1. Each group has a shortcut from its input to the output like Figure 3.4. The rest blocks follow the same rule. The final block is a fully connected layer. The total convolutional layer number is $1 + 3 * 3 + 4 * 3 + 6 * 3 + 3 * 3 + 1 = 50$. Thus, we name it ResNet-50. Similar to the VGG implementation in Faster R-CNN, we won't connect the fully connected layer with the convolutional block yet. ResNet often combines the Feature Pyramid Network (FPN) to increase the scale robustness. Both ResNet and VGG can output the required feature maps for the RPN.

We use the Reginal Proposed Networks (RPN) to suggest the region proposals based on the feature map in the Faster RCNN. The conventional region proposal method used by R-CNN is computationally expensive. The RPN leads to a qualitative change in detection time. Figure 3.5 shows the structure of the RPN. We use a sliding window on the feature map. Then we send the output of the sliding window to an intermediate layer to decrease the dimension. Then the intermediate layer outputs to two fully connected networks. One is used to give classification, and the other is used to give bounding box location (regression). Assume the maximum possible proposals in each sliding window is k. The classification layer gives $2k$ scores, i.e., this proposal is an object or not. The regression layer gives $4k$ bounding box locations $[x, y, h, w]$. h is the height,

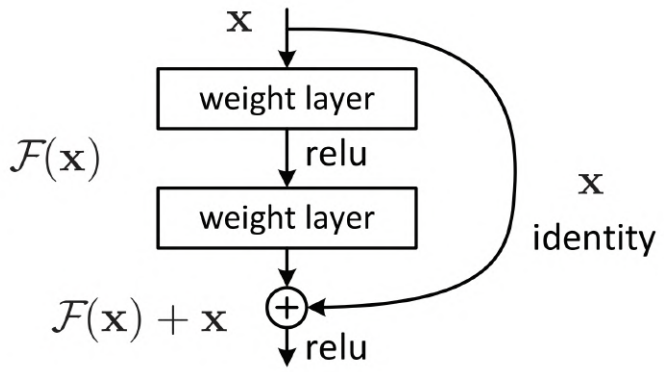


Figure 3.4: ResNet "shorts" the deep layer with the shallow layer.

w is the width. Combining the result of the classification layer and the regression layer, we have object detection.

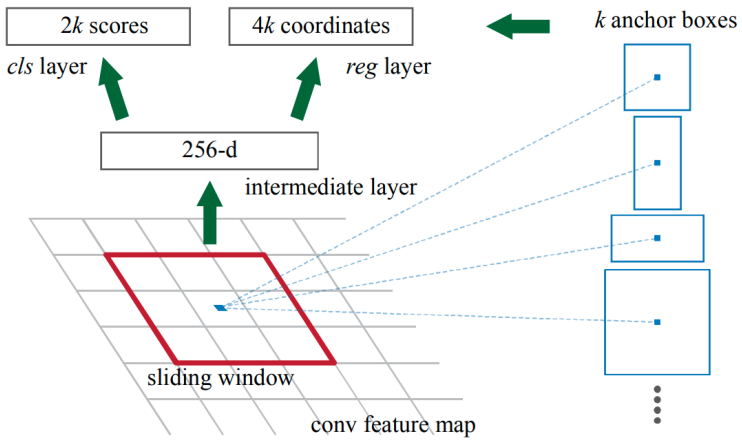


Figure 3.5: Reginal Proposal network classifies the object and gives object bounding box coordinate. k stands for the number of maximum possible proposals. The classification layer gives $2k$ scores, i.e., this proposal is an object or not. The regression layer gives $4k$ bounding box locations $[x, y, h, w]$.

Mask R-CNN combines detection from Faster-R-CNN with the mask. The mask means we have pixel-wise classification instead of just a bool classification in a bounding box. We could perform pixel-wise segmentation by upsampling the backbone output to the same size as the input image. Similar to the Faster RCNN, it uses the ResNet-FPN as its backbone. This method scales the image and extracts a feature map at each layer. This way, we can increase the robustness of the scale invariance. Figure 3.7 [2] shows the structure of the ResNet-FPN. Compared to the ResNet, it has upsampled layers. At the same time, each downsample layer is connected to the upsample layer. Each FPN layer generates a feature map, so we have a feature pyramid. We need to choose ROI-layer correspondence. We use the Eq. (3.1) to make a decision. Where $L_0 = 4$, 224 is image size in the ImageNet for pre-training. If the ROI size width (w) and height (h) are large, 224*225 for example, then we have $L = 5$, which means we should choose the ROI from a higher level (feature map with smaller size). On the other side, if the ROI size is 40*40, we have $L = 3$, which means we should feature a map from layer 3. This method is straightforward because we want the

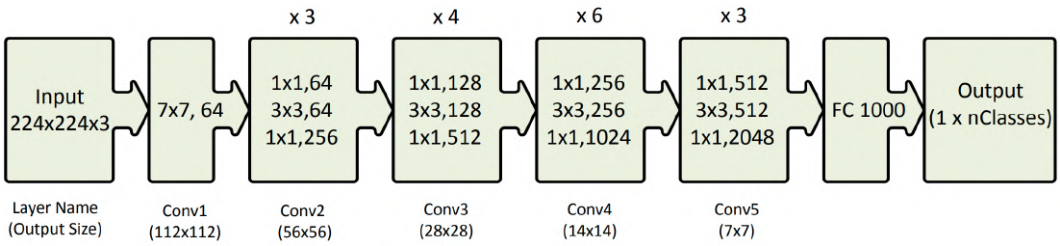


Figure 3.6: ResNet-50 has several blocks, and each block contains a certain number of convolutional layer groups. Figure 3.4 is just one convolutional layer group. There is a shortcut in each layer group. Each group includes several convolutional layers.

large ROI on a "coarse" level while the small ROI on a "precise" level.

$$L = \lfloor L_0 + lg(\sqrt{wh}/224) \rfloor \tag{3.1}$$

Another significant improvement from the Mask RCNN is the "RoIAlign". RoIAlign is an improvement of the "RoIPooling". The RoI Pooling is used to choose features on the feature map in the Faster RCNN. RoI Pooling uses the region proposals to choose a certain position $[x0, y0, h, w]$ on the feature map. Then cut the region into $k * k$ smaller region and choose the maximum number in the feature map. The proposed region $[x0, y0, h, w]$ from the RPN are float numbers. We need to quantize the float numbers and convert them to integers to align the proposed region with the feature map. What's more, we need to cut the proposed region into $k * k$ parts, and the result sub-region may not be an integer, which means we need to quantize it again. Figure 3.8 [5] shows the two steps. The RoI Pooling doesn't have a significant influence on the result of the bounding box chosen. The bounding box is not sensitive to small displacement. However, for the pixel-wise segmentation, RoI Pooling leads to several misalignments between the proposed region and the final mask. The RoIAlign doesn't quantize the proposed region. It uses bilinear interpolation to decide the pixel for max pooling. Figure 3.9 [51] illustrates the RoAlign process. If we want to get the pixel value of the gray circle, then we use bilinear interpolation of the four surrounded pixels (four blue arrows) to calculate the target intensity. We use the same method to get each gray circle's pixel intensity, and finally, we use max-pooling to choose the maximum intensity pixel.

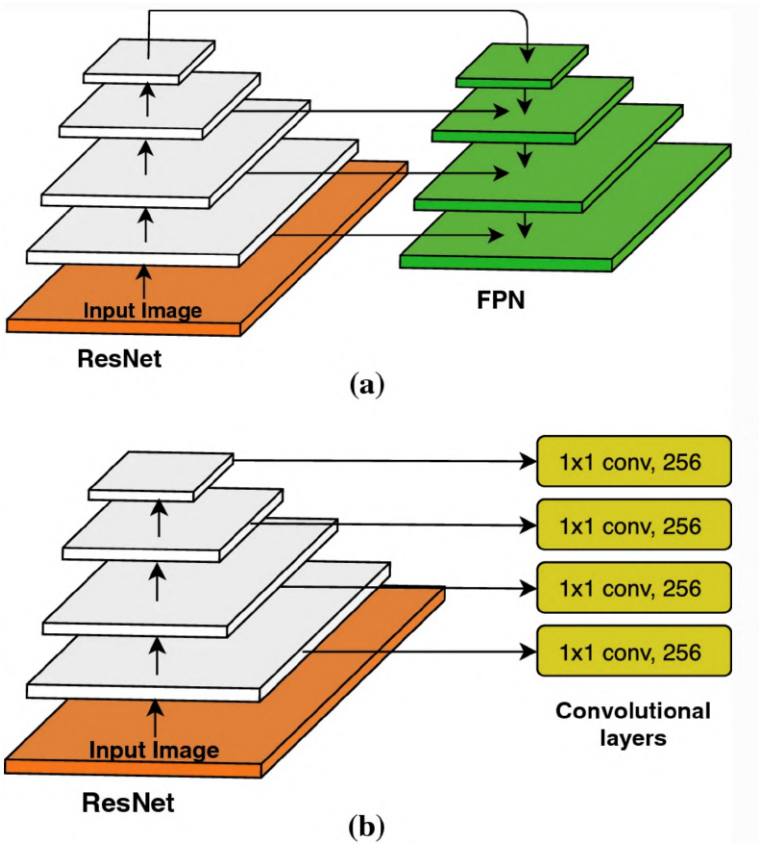


Figure 3.7: ResNet FPN shorts the downsample layer with upsample layer.

The Loss function of the Mask RCNN is as the following

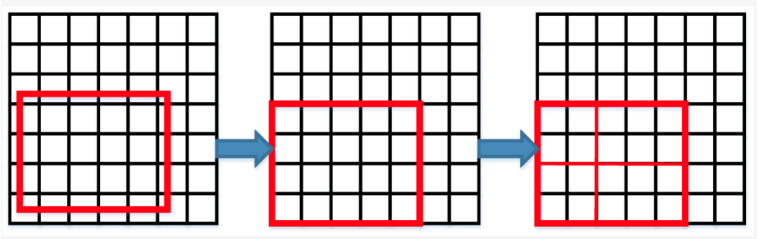


Figure 3.8: The black and white squares are feature maps. The square is the proposed region. RoI Pooling quantizes the RoI and aligns it with the feature map, then it cuts the RoI into sub-regions and uses max pooling for each region.

$$L = L_{cls} + L_{box} + L_{mask} \tag{3.2}$$

where the L_{cls} is the classification loss (object or not), L_{box} is the bounding box loss, L_{mask} is the pixelwise mask loss. For the RoI has $m * m$ pixels, the mask loss has $K * m * m$ terms for K classes.

From the above description, we have a basic understanding of the "R-CNN" family. They have made more improvements in recent years. However, the regional proposals method makes them have the same bottleneck on detection speed. Is it possible to accelerate the detection stage so that the overall masking becomes faster?

The Yolo (You only look once) comes to the stage with a very different basis. Yolo split the input image into S by S grids, and then in each grid, it generates B bounding boxes. For each bounding box, the network will create a class probability and the bounding box location. When the probability is higher than a threshold, the bounding box is chosen, and the object inside is shown. Figure 3.10 shows the simplified Yolo detection process. Yolo can achieve more than 40 fps image detection when it launches. However, there are also shortcomings, Yolo uses a grid approach, and the grid typically is large so that small objects sometimes are ignored. The following Yolo version [88, 89, 15] improve detection speed as well as accuracy.

An image segmentation network Yolact [17, 16] is proposed latter. According to its report, Yolact can achieve 33.5 FPS speed on Nvidia Titan Xp GPU. Figure 3.11 [17] shows its structure.

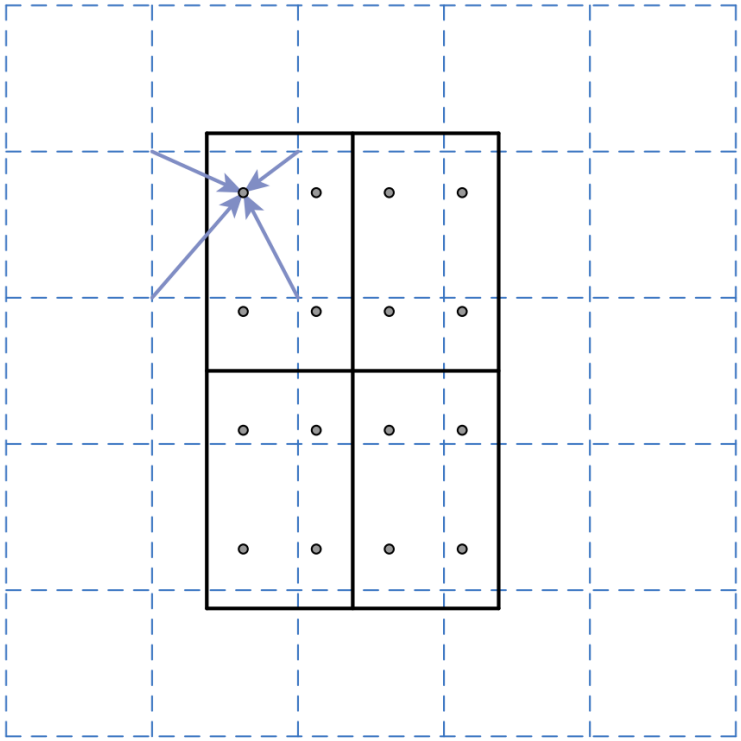


Figure 3.9: RoIAlign doesn't quantize the region proposals from the RPN. Instead, it keeps the float number of the region proposals and directly uses bilinear interpolation to find the pixel for the max pooling.

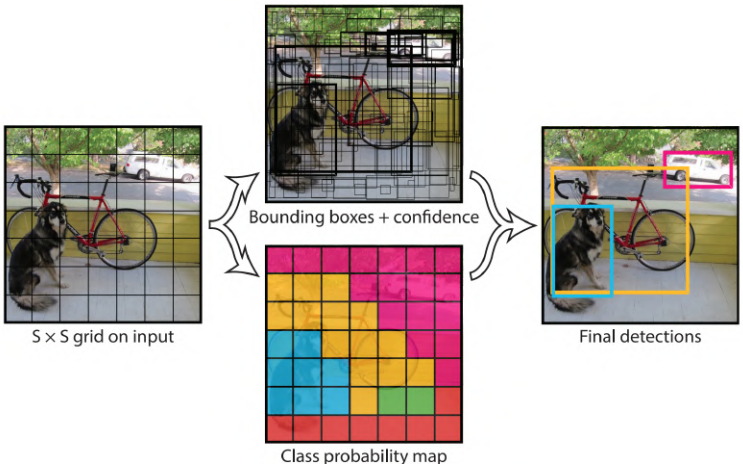


Figure 3.10: The image is split into grids and grid has the prediction confidence of a certain object.

From its structure, we can see its backbone is similar to the Mask RCNN. Both of them use ResNet-FPN. The Yolact, however, doesn't use RoIPooling or RoIAlign. Instead, Yolact uses two parallel branches to help to generate the mask. One branch produces "prototypes" while the other "mask coefficients." Each prototype has a corresponding mask coefficient. Yolact uses a simple linear combination to join the prototype with its mask coefficient and decide the final mask. Yolact mAP is 34.1, according to its report. mAP(mean Average Precision) leverages the precision (percentage of correct prediction vs. all prediction) and recall (percentage of correct prediction vs. all positive cases) of the image detector. A higher mAP indicates a better segmentation.

In our dense reconstruction solution, we use the image segmentation system to decide the

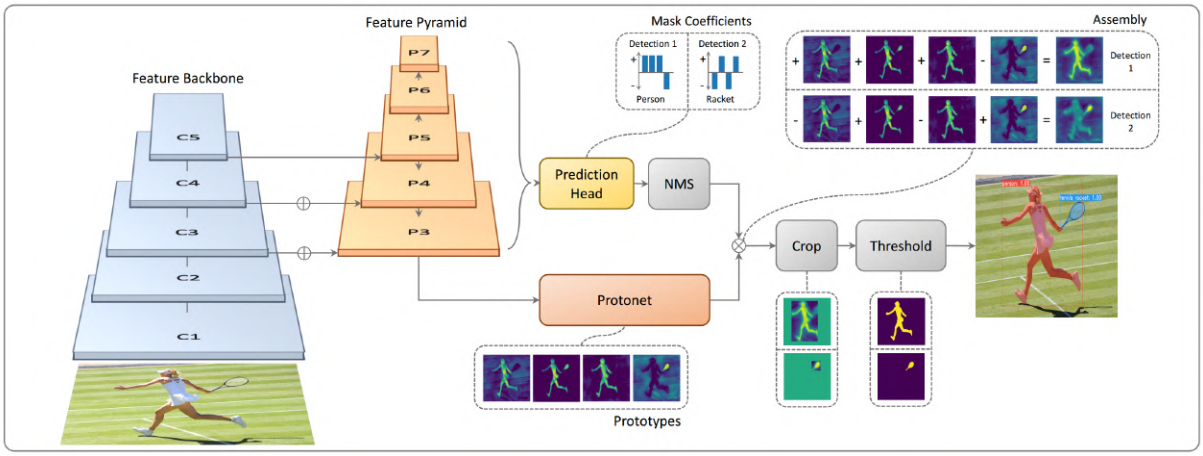


Figure 3.11: Yolact uses ResNet-FPN in the first stage as the backbone, which is similar to the Mask-RCNN. However, it doesn't include RoI pooling or RoI Align at the second stage. It uses mask coefficient combined with prototypes to decide the final mask.

potential dynamic objects. We chose the Yolact because it has a faster speed and a better mask quality than the Mask RCNN. The mask quality indicates if the mask is closely attached to the desired object. We run the Yolact and MaskRCNN with the same images and draw their masks. In Figure 3.12, the red circles show the mask quality difference between MaskRCNN. The images come from different resources such as TUM RGB-D dataset [108], Oxford Robot Car [71], [72], COCO [66] and our customized data.

Figure 3.12: The left side is the detection from the MaskRCNN, and the right side is the detection from Yolact. The red circles show the Yolack has better mask quality than the Mask RCNN. The Yolact misses a person detection in the fourth image. The phenomenon can be relieved by decreasing the detection threshold.

3.0.1 Implementation of Segmentation

Although Mask RCNN and Yolact are the state-of-the-art instance segmentation system, we tend to use a more classical and straightforward network to demonstrate the process of segmentation. In this sub-section, we are going to show the U-Net [92]. U-Net can do semantic segmentation instead of instance segmentation, which means the object belonging to the same class shares the same mask. However, as a toy implementation, it can give us an idea about how the whole image segmentation works. The U-Net names after its "U" structure. "U" structure downsamples and upsamples the image so that we can have pixel-wise segmentation. As we can see, we need this approach more or less for all the pixel-wise segmentation networks. The so-called fully convolutional network [68].

In the downsampling process, the input has five levels. Each level has three convolutional networks followed by ReLU. For example, if the input image has size $572 \times 572 \times 1$, after two convolutional network we have feature map $568 \times 568 \times 64$ because the convolutional core size is 3×3. The output channel is 64. After one layer, we use 2×2 max pooling to downsample the image. Then we re-do the above process several times until the feature map size turns to 32×32 with 1024 channels. Then we upsample the feature map. During the upsample process, again, we use two convolutional networks followed by ReLU in each layer. We use bilinear interpolation to upsample the feature map. We re-do the above process until the output feature map has the same size as the input. The final output channel numbers match with the class number we want to classify. One of the most significant features of the U-Net is it shorts the output and the input layer. There is a blue arrow between each output and the input layer. This idea is similar to the ResNet.

Researchers have implemented the U-Net on various platforms such as Caffe, Tensorflow, and Pytorch. We use Pytorch for implementation, inspired by the previous work [1] . The earlier work only can classify one class and cannot process customized datasets. We improve it on the generality such that it can classify multiple classes and process customized datasets. The code is available [2]

[1] `https://github.com/milesial/Pytorch-UNet`

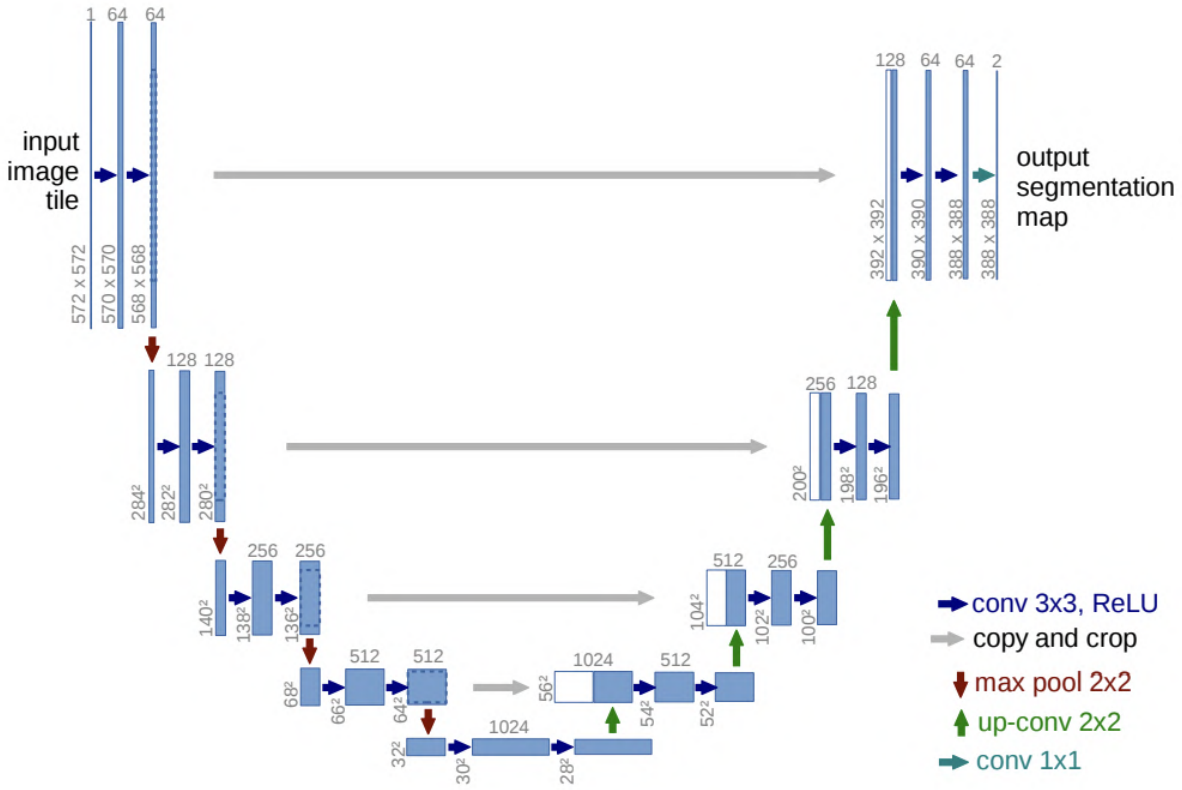


Figure 3.13: In the U-Net, we firstly downsample the image. Each downsample layer has two convolutional networks followed by ReLU. The max-pooling downsamples the feature map. Each input layer connects to the output layer.

. The code is pretty striaghtforward. Here we only shows its core structure.

```
class UNet(nn.Module):
    def __init__(self,n_channels, n_classes, bilinear=True):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes
        self.bilinear = bilinear

        self.inp = DoubleConv(n_channels,64)
        self.down1 = Down(64, 128)
        self.down2 = Down(128,256)
        self.down3 = Down(256,512)
        factor = 2 if bilinear else 1
        self.down4 = Down(512,1024//factor)
        self.up1 = Up(1024,512//factor, bilinear)
        self.up2 = Up(512,256//factor, bilinear)
        self.up3 = Up(256,128//factor, bilinear)
        self.up4 = Up(128,64)
        self.outc = Out(64,n_classes)

    def forward(self, x):
        x0 = self.inp(x)
        x1 = self.down1(x0)
        x2 = self.down2(x1)
        x3 = self.down3(x2)
        x4 = self.down4(x3)
        x = self.up1(x4,x3)
        x = self.up2(x,x2)
        x = self.up3(x,x1)
        x = self.up4(x,x0)
```

[2] https://github.com/zhaozhongch/Pytorch_UNET_MultiObjects

```
        return self.outc(x)
```

In the "U-NET" class, we define member function for different levels. We can see in the constructor function, we input channel and output classes. Then we define each "down" and "up" layers. Each layer contains double convolutional networks, two ReLU, and one max pooling. The details are shown in the link. In the "forward" function, we have three downsamples followed by three upsamples, as shown in Fig. 3.13. We demonstrate its coarse training result. Although it doesn't have a high-quality mask compared to the groundtruth, we think it should be enough for a toy image segmentation system.

Figure 3.14: U-Net segmentation result without fine tuning.

Chapter 4

Object Tracking and Filter Tuning

In this chapter, we discuss the fundamentals of multiple object tracking and our approach for the Kalman filter tuning used in the object tracking. If a 3D reconstruction of dynamic objects, it is natural to think about another question. Can we track them? Tracking means each object has its index, and it is stable even the object moves or the environment changes. Figure 4.4 shows an example of tracking. Object tracking is significant for the task such as obstacle avoidance. Researchers have investigated this area in recent years. Earlier work such as Sort [13] (in fact not that early) builds the basis of the state of the art tracking algorithm, the following work such as Deep-Sort [119], MOTDT [67], and JDE [114] aims at increasing the fps as well as the accuracy. We hope to track the 3D object in the scene according to the 2D object detection and segmentation. Naively there are two main approaches. The first one is we can build a 3D object model and track the 3D model, and the second one is we follow the 2D object and project it to 3D space according to the depth. The tracking in 2D is widely researched because 2D detection work is more developed. The tools such as Yolo and Faster-R-CNN we mentioned in section 3 can all be used for the detection task. Then tracking is to decide if objects detected in different frames are the same objects. Figure 4.1 shows the problem we need to solve. In this thesis tracking part, we'll focus on Multiple Object Tracking (MOT)

Sort [13] builds the foundation of the current state-of-the-art tracking system. Two parts construct Sort: Kalman filter and Hungarian Algorithm [60]. Sort doesn't include any detection system. We need to provide an image and text file with the object detected. Then for the objects

Figure 4.1: We detect two persons in each image. How could we know if a person in one image matches anyone in the other image?

.

detected, Sort initializes a Kalman filter to predict where this object will be. The Kalman filter needs a process model and measurement model. The process model is typically the constant velocity model and the measurement model coming from image detection. The process model tries to track the bounding box instead of the object itself. According to the current bounding box location and constant velocity model, the Kalman filter predicts the bounding box in the next frame. Then if the next frame has several bounding boxes for diverse objects, we need to decide the bijection between the predicted bounding box from the process model and the object. Hungarian Algorithm is used to solve the data association problem. For each bounding box indicated by the Kalman filter, we'll calculate its Intersection over Union (IoU) with each of the bounding boxes in the new frame. We can see the IoU of two bounding box's definition in Figure 4.2. [34].

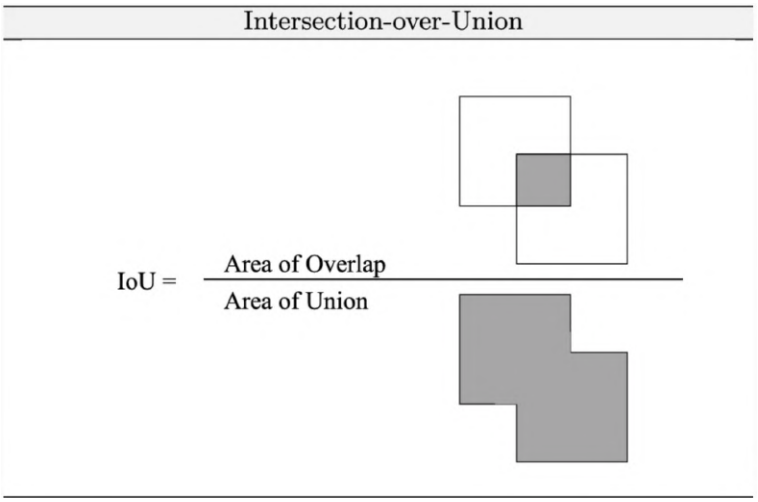


Figure 4.2

If we have m bounding boxes from the Kalman filter process model and n bounding boxes from the next frame's detection, after the above process, we'll end up with a m by n matrix M. The goal of the Hungarian Algorithm is to find a combination that can maximize the IoU sum of each row.

$$\max M(i, j) \tag{4.1}$$

Note that the constraint is that we can only pick one column for each row (each bounding

box from the Kalman filter can only correspond to one bounding box from the following frame). $m > n$ means some objects in the last frame cannot find a correspondence in the next frame; then, we delete those objects' tracking. $m < n$ means that we haven't tracked some objects in the next frame, so we'll initialize those undetected objects in the Kalman filter. However, even $m = n$, the bounding box in the last frame may not be able to find a correspondence in the next frame. For example, in the current frame, some objects leave, and in the next frame, some objects come into view from very different positions. Under this condition, We also need to delete some old trackings and initialize new trackings. From this point of view, it is crucial to set an IoU threshold such that when two bounding boxes don't share a large IoU, we don't apply the Hungarian Algorithm to them. The flow chart 4.2 can show the process of Sort. [114, 67, 112].

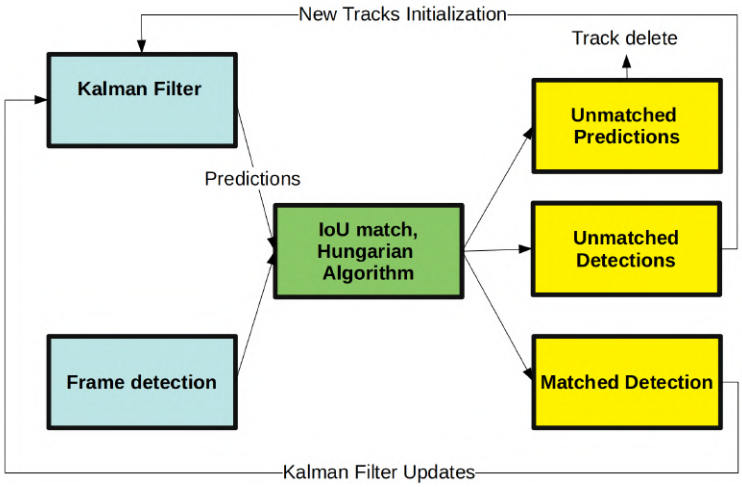


Figure 4.3

Object tracking is a potential application of this work. However, it requires the Kalman filter to estimate the tracking objects' state. We can simply use the constant velocity model as the process model of the Kalman filter. However, it is non-trival to define the noise covariance. We need to *tune* the filter to obtain the noise paramters.

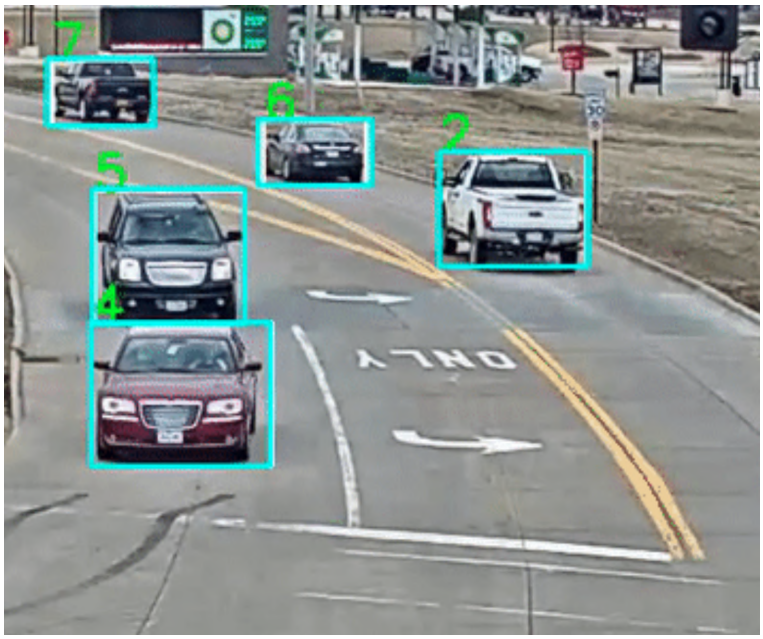


Figure 4.4: Tracking via deep sort. As you can see, each car has an index above its corresponding bounding box. The index won't change when the car moves from place to place

4.1 Kalman Filter Tuning

We define the state estimation problem given the process and measurement model before we dig into the Kalman filter tuning problem.

The state of the system at timestep k is $\mathbf{x}k$. The system is described by continuous time process model and observation models,

$$\begin{aligned} \dot{\mathbf{x}}_t &= \mathbf{A}\mathbf{x}t + \mathbf{G}\mathbf{u}_t + \mathbf{\Gamma}\mathbf{v}_t, \\ \mathbf{z}_t &= \mathbf{H}\mathbf{x}t + \mathbf{w}_t, \end{aligned} \tag{4.2}$$

where $\mathbf{u}_t$ is the control input, the process noise is the additive white process $\mathbf{v}_t$ with intensity $\mathbf{V}$, and the measurement noise is an additive white noise process $\mathbf{w}_t$ with continuous time intensity $\mathbf{W}$. The discrete time system evolution from timestep $k-1$ to k is

$$\mathbf{x}k = \mathbf{F}_k\mathbf{x}k-1 + \mathbf{B}_k\mathbf{u}_k + \mathbf{v}_k, \tag{4.3}$$

where $\mathbf{u}_k$ is the control input and $\mathbf{v}_k$ is the process noise, which is assumed to be zero mean and independent with covariance $\mathbf{Q}_k$. The observation model is

$$\mathbf{z}_k = \mathbf{H}_k\mathbf{x}k + \mathbf{w}_k, \tag{4.4}$$

where $\mathbf{w}_k$ is the observation noise.

The discrete-time system is derived from the continuous time system using techniques such as Van Loan's method [74],

$$\begin{aligned} \mathbf{F}_k &= e^{\mathbf{A}_t\Delta t}, \quad \mathbf{B}_k = \int_0^{\Delta t} e^{\mathbf{A}_t m}\mathrm{d}m, \\ \mathbf{Q}_k &= \int_0^{\Delta t} e^{\mathbf{A}_t m}\mathbf{\Gamma}\mathbf{V}\mathbf{\Gamma}^T e^{\mathbf{A}^T m}\mathrm{d}m. \end{aligned} \tag{4.5}$$

If the observation is from an integrating sensor, the discrete time observation vector is [74]

$$\mathbf{R}k = \frac{\mathbf{W}}{\Delta t}. \tag{4.6}$$

For a non-integrating sensor $\mathbf{R}k = \mathbf{R}t$.

A Kalman filter can be used to find the optimal state estimate [55], via a two stage process

of prediction followed by measurement update. The prediction is

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{F}_k \hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k \mathbf{u}_k \tag{4.7}$$

$$\mathbf{P}_{k|k-1} = \mathbf{F}_k \mathbf{P}_{k-1|k-1} \mathbf{F}_k^\top + \mathbf{Q}_k \tag{4.8}$$

while the update is

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + \mathbf{W}_k \mathbf{e}_{\mathbf{z},k}, \tag{4.9}$$

$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - \mathbf{W}_k \mathbf{S}_{k|k-1} \mathbf{W}_k^\top, \tag{4.10}$$

$$\mathbf{S}_{k|k-1} = \mathbf{H}_k \mathbf{P}_{k|k-1} \mathbf{H}_k^\top + \mathbf{R}k \tag{4.11}$$

$$\mathbf{W}_k = \mathbf{P}_{k|k-1} \mathbf{H}_k^\top \mathbf{S}_{k|k-1}^{-1} \tag{4.12}$$

where $\mathbf{e}_{\mathbf{z},k} = \hat{\mathbf{z}}_{k|k-1} - \mathbf{z}_k$ is the innovation vector.

The general idea behind the Kalman filter tuning is to find a cost function $\mathcal{F}$ to indicate the performance of the Kalman fitler and use an optimizer $\mathcal{O}$ to minimize the cost function w.r.t the noise parameter.

Two widely used measures for fitness are the *Normalized estimation error squared (NEES)* and the *normalized innovation error squared (NIS)*. The NEES is computed from

$$\epsilon_{\mathbf{x},k} = \mathbf{e}_{\mathbf{x},k}^T \mathbf{P}_{k|k}^{-1} \mathbf{e}_{\mathbf{x},k}, \tag{4.13}$$

where $\mathbf{e}_{\mathbf{x},k} = \hat{\mathbf{x}}_{k|k} - \mathbf{x}k$. Because we need to know $\mathbf{x}k$, NEES requires a groundtruth measurement of the system state. The NIS, on the other hand, only depends on the observation sequence and does not require knowledge of groundtruth. It is computed from

$$\epsilon_{\mathbf{z},k} = \mathbf{e}_{\mathbf{z},k}^T \mathbf{S}_{k|k-1}^{-1} \mathbf{e}_{\mathbf{z},k}, \tag{4.14}$$

where $\mathbf{e}_{\mathbf{z},k}$ is the innovation vector. If the filter is statistically consistent, it can be shown that the expected values of the NEES and the NIS are [7]

$$\mathrm{E}\left[\epsilon_{\mathbf{x},k}\right] \approx n_{\mathbf{x}}, \quad \mathrm{E}\left[\epsilon_{\mathbf{z},k}\right] \approx n_{\mathbf{z}}, \tag{4.15}$$

Although the $\epsilon_{\mathbf{z},k}$ and $\epsilon_{\mathbf{x},k}$ are widely used, they have the property that they are bounded from below (by 0) but not from above. This naturally introduces a bias or asymmetry in the measure. To overcome this, we use a log measure instead:

$$\begin{aligned} \mathcal{F} \doteq J_{NEES} &= \left| \log \left(\frac{\sum_{k=1}^{T} \bar{\epsilon}_{\mathbf{x},k}/T}{n_{\mathbf{x}}} \right) \right|, \\ \bar{\epsilon}_{\mathbf{x},k} &= \frac{1}{N} \sum_{i=1}^{N} \epsilon_{\mathbf{x},k}^{i}. \end{aligned} \tag{4.16}$$

where N is the number of Monte Carlo runs and T is the period of sampling. Note that this cost function is not bounded.

We use the Bayesian Optimization (BO) as our optimizer $\mathcal{O}$. BO poses optimization as a Bayesian search problem. The objective function is unknown and is treated as a random variable. A prior is placed over it. As the algorithm proceeds, each iteration takes samples from the objective function which are used to refine the distribution. The next sample point is selected to maximize the probability of improving the current best estimate. In short, We use BO for the generic 'black box' stochastic objective functions.

Consider the minimization of some objective function $y : \mathcal{Q} \to \mathbb{R}$, where $\mathcal{Q} \in \mathbb{R}^d$ is the search or solution space, and the element $\mathbf{q}^* \in \mathcal{Q}$ is the minimizer, such that $y(\mathcal{Q}^*) \leq y(\mathcal{Q}),\ \forall \mathbf{q} \in \mathcal{Q}$. Bayesian optimization uses black box point evaluations of y to efficiently find $\mathbf{q}^*$. This is accomplished by maintaining beliefs about how y behaves over all $\mathbf{q}$ in the form of a "surrogate model" $\mathcal{S}$, which statistically approximates y and is easier to evaluate (e.g. since y might be an expensive high fidelity simulation). During optimization, $\mathcal{S}$ is used to determine where the next design point sample evaluation of y should occur, in order to update beliefs over y and thus simultaneously improve $\mathcal{S}$ while finding the (expected) minimum of y as quickly as possible. The key idea is that, as more observations are sampled at different $\mathbf{q}$ locations, the $\mathbf{q}$ samples themselves eventually converge to the expected minimizer $\mathbf{q}^*$ of y. Since $\mathcal{S}$ contains statistical information about the level of uncertainty in y (i.e. related to $p(y|E)$, the posterior belief), Bayesian optimization effectively leverages probabilistic 'explore-exploit' behavior to learn an approximate model of y while also minimizing it. There are two main components of the Bayesian optimization

process: (1) the surrogate model $\mathcal{S}$, which encodes statistical beliefs about y in light of previous observations and a prior belief; and (2) the acquisition function $a(\mathbf{q})$, which is used to intelligently guide the search for $\mathbf{q}^*$ via $\mathcal{S}$. Figure 4.5 shows an example of surrogate model.

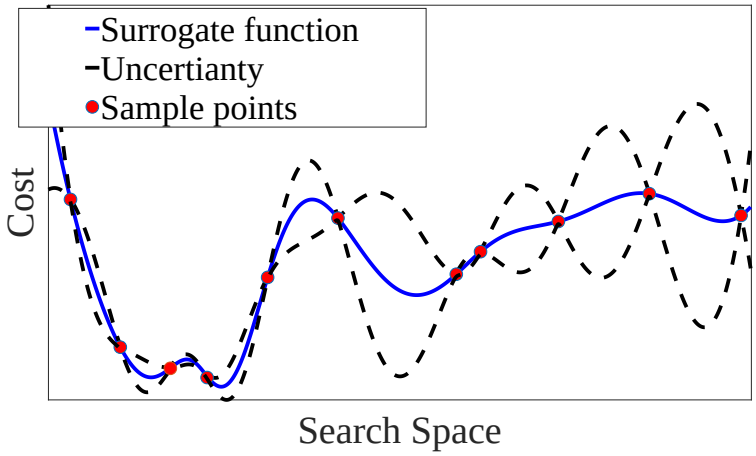


Figure 4.5: Example of the 1D surrogate model of the BO. BO uses Gaussian Process as the surrogate model. It naturally gives the uncertainty of the "guess". The dashed line is the uncertainty, blue line is the surrogate model, red dots are the sample points from the objective function.

Surrogate model $\mathcal{S}$ must approximate y in areas where it has not yet been evaluated, and must also provide a predicted value and corresponding uncertainty to quantify the possibility that the optimum is located at some location $\mathbf{q}$. Gaussian Processes (GPs) [86] are the most common family of surrogate models used in Bayesian optimization; the acronym GPBO here refers to Bayesian optimization using a GP surrogate model $\mathcal{S}$. A GP describes a distribution over functions; it is more formally defined as a collection of random variables, any finite number of which have a joint Gaussian distribution [14, 86],

$$f(\mathbf{q}) \sim \mathcal{GP}(m(\mathbf{q}), k(\mathbf{q}, \mathbf{q}')) \tag{4.17}$$

$$m(\mathbf{q}) = \mathbb{E}[f(\mathbf{q})] \tag{4.18}$$

$$k(\mathbf{q}, \mathbf{q}') = \mathbb{E}[(f(\mathbf{q}) - m(\mathbf{q}))(f(\mathbf{q}') - m(\mathbf{q}'))] \tag{4.19}$$

where the process is completely specified by its mean function $m(\mathbf{q})$ (Eq. 4.18), and its covariance function $k(\mathbf{q}, \mathbf{q}')$ (Eq. 4.19). In theory $m(\mathbf{q})$ could be any function; as is common practice, this work

assumes m is zero for simplicity. The covariance (or kernel) function is a mapping $k : (\mathbf{q}, \mathbf{q}') \to \mathbb{R}$; this must be specified a priori, and is usually based on some knowledge of y's smoothness properties. In Figure 4.5, we obtain the uncertainty from the kernel function.

The other main component of the BO is the acquistion function. The acquisition function is defined as the mapping $a : (\mathbf{q}, \mathcal{S}) \to \mathbb{R}$, abbreviated as

$$a(\mathbf{q}) \triangleq a(\mathbf{q}, \mathcal{GP}(m(\mathbf{q}), k(\mathbf{q}, \mathbf{q}'))) \tag{4.20}$$

which assumes the inclusion of the GP surrogate model as an argument. GPBO selects $\hat{\mathbf{q}} = \text{argmax}_{\mathcal{Q}}\, a(\mathbf{q})$ as the next location in $\mathcal{Q}$ to be evaluated in the search process. Ideally, $a(\mathbf{q})$ should enable exploration and modeling of y by sampling new locations $\mathbf{q}$ that will improve the accuracy of $\mathcal{S}$. At the same time, $a(\mathbf{q})$ must exploit $\mathcal{S}$ to reach the expected minimum of y as quickly as possible. Therefore, $a(\mathbf{q})$ should not lead to greedy or myopic behavior, or get stuck in poor local minima. There are many ways to define $a(\mathbf{q})$ to balance these needs, but the best choice is heavily application dependent [101, 18, 53]. Some popular methods include Expected Improvement (EI) and the Upper Confidence Bound. We describe the details of the surrogate model and acquistion function in our previous work [22], [23].

We start with a toy simulation to visualize BO tunning process. Consider a robot that moves along a 1D track and receives position measurements every $\Delta t = 0.1s$. Suppose the position and velocity state $\mathbf{x} = [\xi, \dot{\xi}]^T$ are governed by the linear time invariant kinematics model

$$\dot{\mathbf{x}}_t = \mathbf{A}\mathbf{x}t + \mathbf{G}\mathbf{u}_t + \mathbf{\Gamma}\mathbf{v}_t$$

$$\mathbf{z}_t = \mathbf{H}\mathbf{x}t + \mathbf{w}_t,$$

where

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad \mathbf{\Gamma} = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

and the inputs to the system consist of a control acceleration $\mathbf{u}_t$, additive white Gaussian noise acceleration process $\mathbf{v}_t$ with intensity $\mathbf{V}$, and additive white Gaussian position measurement noise

process $\mathbf{w}_t$ with continuous time intensity $\mathbf{W}$. The control input $\mathbf{u}_t = 2\cos(0.75t)$ causes the robot to move with a low frequency oscillation. Applying a zero-order hold discretization to this system, we can obtain discrete time position and velocity state $\mathbf{x}k = [\xi_k, \dot{\xi}_k]^T$. We set the groundtruth noise parameter $\mathbf{V} = 1, \mathbf{W} = 0.1$. But we run the Kalman filter with different noise parameter value. For each pair of $(\mathbf{V}, \mathbf{W})$ noise we obtain the cost from $\mathcal{F}$. Thus, we can have a 3D $\{\mathbf{V}, \mathbf{W}, \mathcal{F}\}$ view. In Figure 4.6, we show the bird view of this 3D plot. This figure helps us to know the true objective function. The red arrow points to the minimal value, which is the groundtruth $(\mathbf{V}, \mathbf{W})$

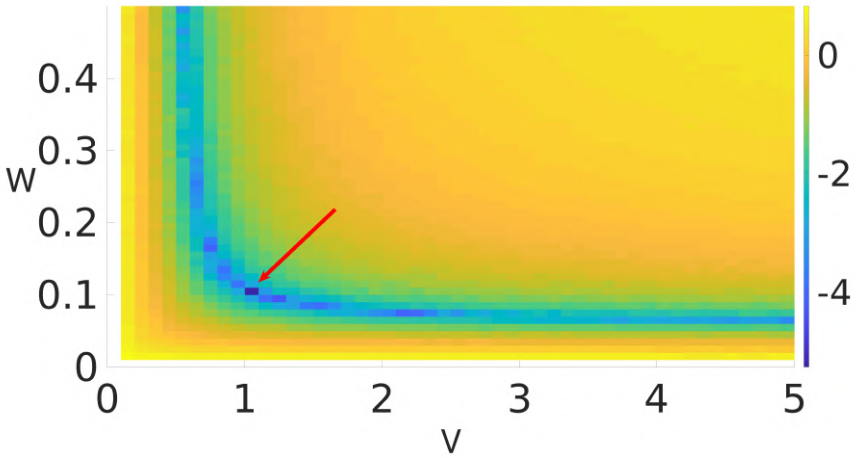


Figure 4.6

Next, we run the BO and assume the objective function is unknown. As we discussed in this chapter, the BO tries to "guess" the objective function according to the sample. In Figure 4.7 we demonstrate the example of the BO search. We run the BO to optimize the Kalman filter with 200 iterations and 40 initial samples. In the Sub-Figure 4.7a we show the initial surrogate model after the 40 initial samples. The surrogate model $\mathcal{S}$ at the current stage is pretty different from the "true" objective function Figure 4.6. However, as we have more and more samples, we can see from the Sub-Figure 4.7a to 4.7d the $\mathcal{S}$ is almost identical to the objective function. At the same time, the BO has more samples around the groundtruth value $(\mathbf{V} = 1, \mathbf{W} = 0.1)$ place. From the above

visualization, we can understand the BO is a practical tool to optimize a "black box" stochastic function. It can provide a global optimization result, uncertainty, and vivid visualization. We release the code of this project according to our previous work [1] .

In this chapter, we talk about multiple object tracking and its essentials. The multiple ob-

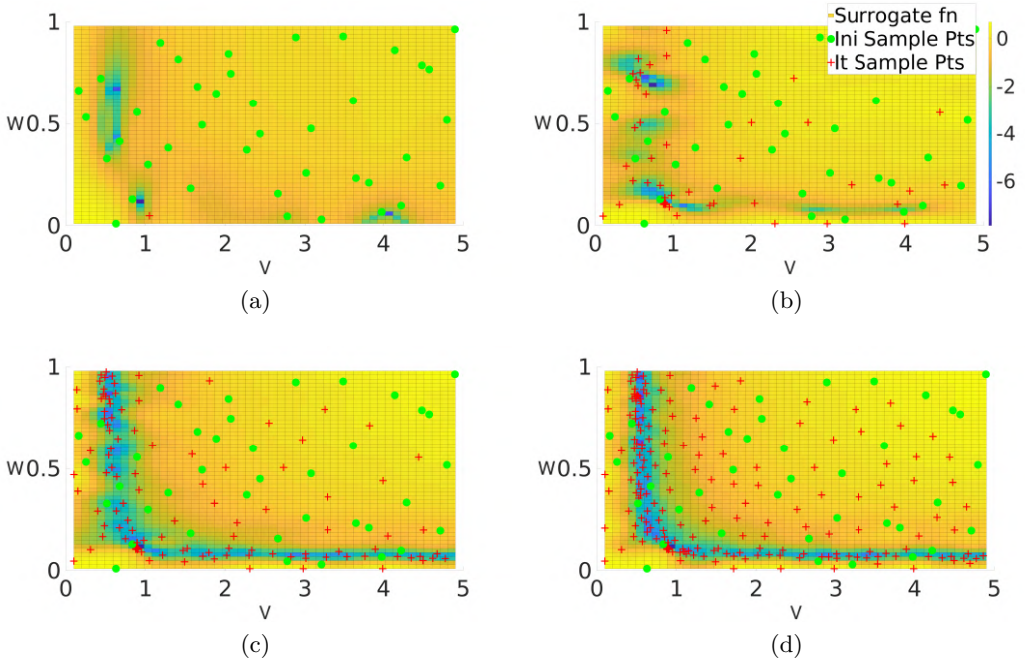


Figure 4.7: GPBO surrogate model for J_{NEES} cost, showing initial random sample points (green dots) and estimations (red crosses) infered by BO in different iterations. From (a) to (d), with more and more estimations, our algorithm successfully explore the cost space. The final surrogate model is similar to the real cost surface from Figure 4.6. Finally, it finds the minimum around $\mathbf{V} = 1$ and $\mathbf{W} = 0.1$.

ject tracking needs object detection and the Kalman filter to estimate the predictable object states. Besides the model, one of the most challenging task in the Kalman filter is tuning, and we briefly discuss our tuning work towards the Kalman filter application. Thanks to the Bayesian optimization, we can infer the "black box" stochastic function. The multiple object tracking in the 3D space can be a great potential of our application since we need object detection and reconstruction.

[1] `https://github.com/arpg/kf_bayesopt`

Chapter 5

Point Fusion

In this chapter, we talk about the preliminaries for the dense reconstruction systems and discuss how to implement them into a large scale environment. The state-of-the-art dense SLAM system is usually called "Fusion" because localization sensors and fusing dense points from different frames are equally important. For the general sparse SLAM, the sparse map is used to help the localization. In fact, in sparse visual SLAM, we already have a point fusion task because we use feature-based SLAM as an example; each frame's 2D features need to fuse into a global sparse 3D model for camera tracking. There is no significant difference between the sparse 3D model and the dense 3D model when using points. However, due to the dense SLAM system's considerable points need to fuse, they develop different data structures to store the points. Besides directly saving the global 3D model as points, there are two other main data structures.

5.1 Surfel

Elastic fusion [118] is a representative using surfel structure for dense reconstruction. It stores an unordered list of surfels. Each surfel $\mathcal{M}$ has the following attributes: a position $\mathbf{p} \in \mathbb{R}^3$, normal $\mathbf{n} \in \mathbb{R}^3$, color $\mathbf{c} \in \mathbb{N}^3$, weight $\omega \in \mathbb{R}$, radius $r \in \mathbb{R}$, initialisation timestamp t_0 and last updated timestamp t (Note not all surfel based dense visual SLAM has the same data structure but should be similar [120, 99, 107, 81]). The new frame generates a set of point clouds, and they are fused into surfels. The 3D geometry is generally constructed according to the position and the color of the surfel. The elastic fusion is designed for a small environment reconstruction and can

get a pretty precise result. The picture 5.1 from the elastic fusion includes 4.5 million surfels and takes about 250 MB of memory for a small office room. A large space will increase memory usage a lot and decrease fusion speed.

Figure 5.1: Elastic Fusion's reconstruction for a office room.4.5 million surfels are included

5.2 TSDF

TSDF (Truncated Signed Distance Function) is another popular approach to render the 3D model and is applied by many dense reconstruction applications [76, 20, 79, 54, 40]. In the TSDF application, the 3D world is divided into a large number of voxels. Each voxel occupies a 3D cube in space, which is shown in figure 5.2. The voxel is the basic unit of 3D space like the pixel to 2D space. Each voxel stores the following value: TSDF $d \in \mathbb{R}$, weight $\omega \in \mathbb{R}$, color $\mathbf{c} \in \mathbb{N}^3$ (Again, not all the TSDF application has the same basic data structures). TSDF value shows the distance between the voxel center and its closest surface. This property is critical. Figure 5.3 [116] shows a bird view of TSDF calculation. In figure 5.3 (a), The blue square represents the camera, and the blue triangle represents the camera field of view. The green polyline stands for a surface, and $\mathbf{p}$ is a point on the surface. $\mathbf{x}$ is the voxel center. We can see from the camera center, and we cast a blue line to go through $\mathbf{x}$ and $\mathbf{p}$. Then the distance between the $\mathbf{x}$ and $\mathbf{p}$ are the TSDF required. From

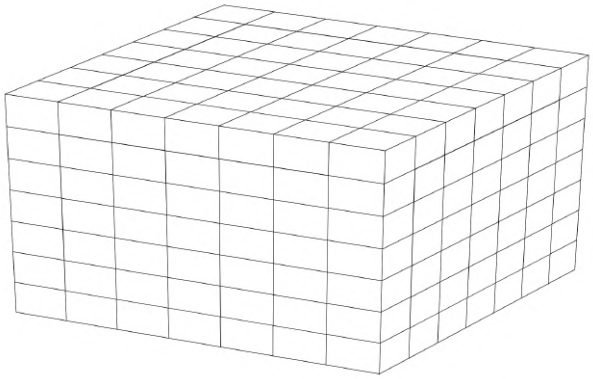


Figure 5.2: Voxel in 3D space

the color bar on the right side, we can see when the voxel is in front of the surface, the distance is a negative value, while when the voxel is at the behind of the surface, it is a positive value, so the TSDF is a "signed" distance. What's more, from figure 5.3 (b), we can see if the voxel is too far away from the camera or too close to the camera, its TSDF is truncated to -1 or 1, that's where the name "truncated" coming from. Truncated value can save computation resources.

We hope to render a surface at the voxel whose TSDF is exactly zero. However, this is normally impossible. Some algorithms try to render a surface when the TSDF is smaller than a threshold of 2cm, which is inaccurate. The marching cube algorithm [69, 25, 78, 77] is designed to decide where to render a surface according to the TSDF value of voxels. In figure 5.4 [29] we demonstrate how the marching cube algorithm works. We chose 8 voxels as a unit. The red circle is a voxel whose TSDF is large than 0 while the others' TSDF is smaller than 0. We interpolate virtual voxels whose TSDF = 0 between the neighbors whose TSDF has a negative sign and render surface there. In the figure, there is just one voxel with TSDF large than 0, and we can interpolate three virtual voxels whose TSDF = 0 and then render a triangle. Each voxel can have TSDF larger than 0 or smaller than 0, so for 8 voxels, we have $2^8 = 256$ possibilities. Among them, there are 15 basic shapes that we can render, which can be seen in figure 5.5 [29]. We can obtain the remaining 241 cases from rotation or translation from the primary cases.

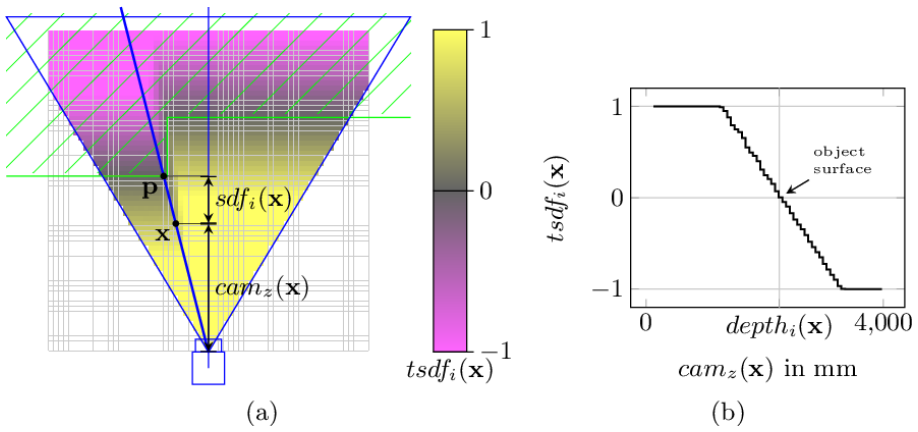


Figure 5.3: TSDF value is the distance between the voxel $\mathbf{x}$ and the closest surface. The distance is obtained by casting a ray from the camera center to the voxel. The first intersection point of the ray to the surface after voxel is $\mathbf{p}$. length $|\mathbf{xp}|$ with sign and truncation is the so-called TSDF.

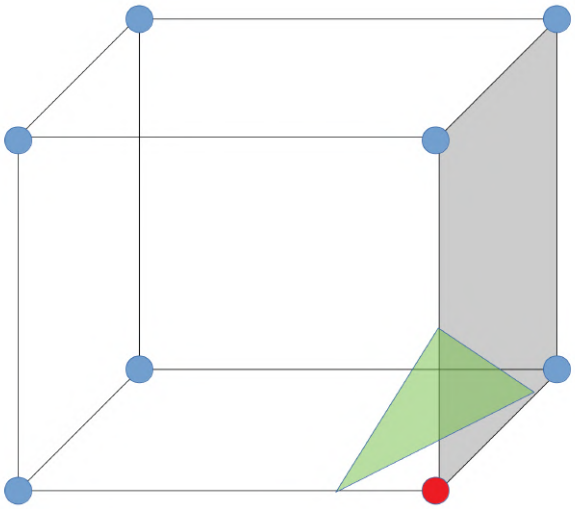


Figure 5.4: We chose 8 voxels as a unit. The red circle is a voxel whose TSDF is larger than 0, while the others' TSDF is smaller than 0. We interpolate virtual voxels whose TSDF = 0 between the neighbor whose TSDF has a negative sign and render surface there.

So far, we have covered TSDF primaries and the method to reconstruct a surface once we have a TSDF value in a voxel. The TSDF value in a voxel is not fixed. When there is a new surface coming, we consider updating the TSDF value. The surface is represented as numerous points from each frame. Each point stands for part of the surface, so when the ray cast from the camera center meets the point, we modify all the TSDF values on the ray. In practice, we normally update all voxels on the ray from the camera center to truncation distance δ behind the point. Some applications are not based on ray casting, but "projection mapping" [76, 57]. This approach updates the TSDF value by projecting the voxel back to the camera frame and comparing the TSDF value with depth. It is fast but leads to strong aliasing for large voxels. We are considering a reconstruction system that can work on a large scale, and setting the voxel size to be a relatively large number is needed, so we plan to use the ray casting approach application in Chapter 6. TSDF update formula is just calculating the average value of all [28]. In equation 5.1, i is the frame id, ω_i is the initial weight of a voxel when updating. Some systems use constant number 1 as ω_i [76] while others determine the weight according to the distance between the point and voxel [80]. Lower case

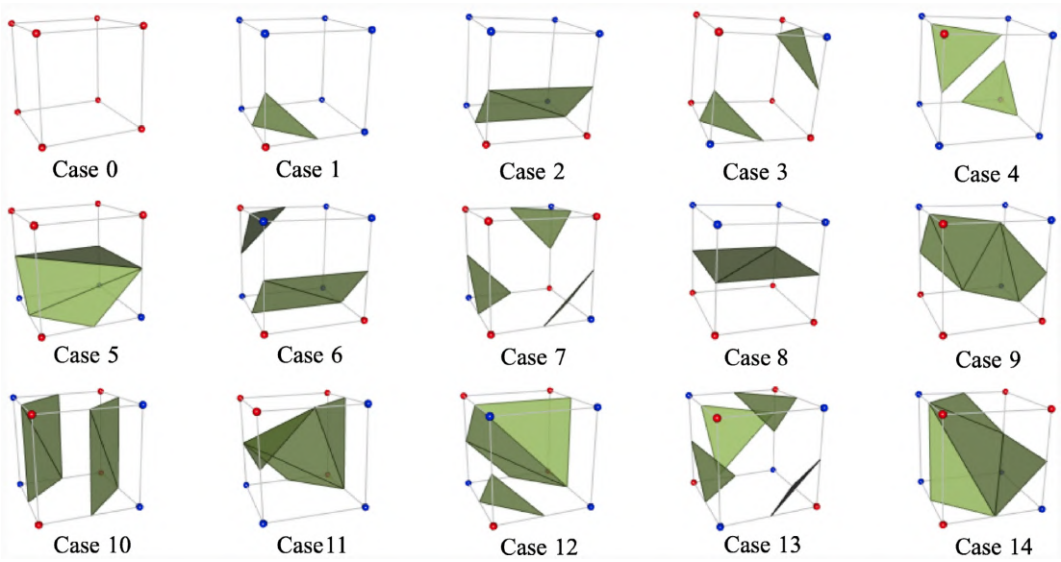


Figure 5.5: 15 basic possibilities rendering surfaces.

d_i is the Euclidean distance between the voxel center and the point considering the truncation and sign requirement.

$$
\begin{aligned}
d_i &= sign(||\mathbf{p} - \mathbf{x}||) \\
D_i(\mathbf{x}) &= \frac{\Sigma\omega_i(\mathbf{x})d_i(\mathbf{x})}{\Sigma\omega_i(\mathbf{x})} \\
W_i(\mathbf{x}) &= \Sigma\omega_i(\mathbf{x})
\end{aligned}
\tag{5.1}
$$

As each frame comes into the system incrementally, so the TSDF update is also incrementally. We rewrite the above equation into equation 5.2. Based on equation 5.2 we can fuse all the incoming points generated by the RGB-D image into the voxels. Then we can use the marching cube algorithm to decide where to insert the surfaces. Lastly, we need to note that the way to decide the number of voxels is hashed [79] because normally, we cannot decide how many voxels we need initially. We need to decide if we need to initialize a new voxel according to a point existence.

$$
\begin{aligned}
d_{i+1} &= sign(||\mathbf{p} - \mathbf{x}||) \\
D_{i+1}(\mathbf{x}) &= \frac{W_i(\mathbf{x})D_i(\mathbf{x}) + \omega_{i+1}(\mathbf{x})d_{i+1}(\mathbf{x})}{W_i(\mathbf{x}) + \omega_{i+1}(\mathbf{x})} \\
W_{i+1}(\mathbf{x}) &= W_i(\mathbf{x}) + \omega_{i+1}(\mathbf{x})
\end{aligned}
\tag{5.2}
$$

5.2.1 Voxel Hashing

Fig 5.2 indicates that we need to allocate space for a voxel in advance. Assume a voxel contains the following structure.

```
struct Voxel {
  float distance = 0.0f;
  float weight = 0.0f;
  Color color;
};
```

where the "Color" contains 4 $uint8$ type variables for RGB color and transparency. One voxel occupies 12 bytes of memory since one $float$ occupies 4 bytes and $Color$ needs 4 bytes. Note when we calculate the struct size, we must consider padding effect [63]. If one voxel represents $1 * 1 * 1cm^3$

real space, then for $1m^3$, we need to use 10^6 voxels, which is $10^6 * 12$ bytes, 11.44MB. An 16GB RAM can only store about 1143 m^3 space. Considering we still need the RAM to store other data, our voxels can only represent limited space. In that case, we cannot reconstruct an ample space. However, when we reconstruct the space using the voxel, we only consider the "surface", which means we have a lot of space that is "free." We should consider allocating the space dynamically, i.e., only giving space where we want to reconstruct the surface. We use the voxel hashing [79] to solve this problem.

We assign each voxel in a $8 \times 8 \times 8$ voxel block. The voxel block has the following structure

```
//Each voxel block maps to a hashentry
struct HashEntry {
  short position[3];
  short offset;
  int pointer;
};
```

The *position* represents the 3D corner position of the block. $position[3] = x, y, z$. We can map the corner position to a hash index by the following equation

$$H(x, y, z) = ((x \times p1) \oplus (y \times p2) \oplus (z \times p3)) \ mod \ K \tag{5.3}$$

where the $p1 = 73856093, p2 = 19349669, p3 = 83492791$ [110]. Note the three numbers are arbitrary, but it has to be a large enough prime number. The K is the hashtable bucket size. $\oplus$ is the XOR operation. In the implementation, we use only bitwise operation for efficiency.

$$H(x, y, z) = ((x \times p1) \oplus (y \times p2) \oplus (z \times p3)) \ \& \ (K - 1) \tag{5.4}$$

where & is the bitwise *and* operation and $K = 2^n, n \in \mathbb{R}$. We can know why the hash function works by answering the following questions.

- Why we need to multiply prime numbers?
- Why we use XOR to combine x, y, z location?

- Why we can use "& $(K-1)$" to replace "*mod* K"?

We use the prime number to avoid hash collisions as much as possible. To accelerate the search, we hope each bucket can contain fewer elements, and each has a similar number of elements. A bad hash function has some buckets "crowded" while some others empty. If we have keys $key = 8, 16, 20, 24, 31$ and 10 buckets, hash function $b = key\%4$, then four elements go to bucket 0 and only 31 goes to bucket 3. However, if we have a hash function $b = key\%7$, every element goes to a special bucket. The prime number only has factor 1 and itself. Thus, a prime number, either in the denominator or in the numerator, can increase the probability that the buckets have comparable elements when using the *mod* operation.

The XOR operation may not be the best way to combine different hash functions but an efficient way. If two numbers have a roughly random distribution, the XOR operation yields a random distribution depending on two values. For two random numbers, we can assume each bit has a 50-50 percent chance of 1 or 0, and then the output still has 50-50 percent change to be 1 or 0. If we use the AND operation, then if one input is, $False$ the output is $False$ no matter what the other input is. For the XOR, both inputs always matter.

The bit operation is typically more efficient than the modulo. The modulo operation may need several clock cycles on the modern CPU architecture, while the bitwise operation typically only needs one clock cycle. We should try to use the bit operation to replace the modulo operation if it is possible. For an unsigned integer N, $N\%K == N\&(K-1)$ if $K = 2^n, n \in \mathbb{R}$.

$$61\%8 \overset{\text{bit operation}}{\rightarrow} 00111101 >> 3 \overset{\text{remove}}{\rightarrow} 00000101$$

$$61\&(8-1) \overset{\text{bit operation}}{\rightarrow} 00111101 \ \& \ 00000111 == 00000101$$

From the above example, we can see that if $K = 2^n$, we move the number N to the right for n digits. The exact n digits are the modulo we want to calculate. The $K-1$ keeps the last n digits as 1, keeping the previous n digits with & operation.

We discussed the first member *position*[3] of the hash entry and the hash function in the above section. The second hash entry is the *offset*. Each bucket only contains s small number of

hash entries. Collision is still inevitable regardless of the hash function in our case. When the collision happens, we find the next hash entry available incrementally and put it into that bucket. When a bucket overflows, we'd like to put the new hash entry into the next bucket. If the next bucket overflows too, we try to find the next of the next bucket. Thus, we utilize the linked list data structure to deal with the overflow buckets. We use the *offset* value to find the linked list. Default *offset* value is zero. If not, it means the entry connects to another element in the linked list. This idea is clearly shown in Fig. 5.6. We also need to use the *offset* to solve another two typical operation of the hash table: *retrieval* and *deletion*.

Retrieval means we want to obtain the hash entry according to a voxel block position. The first step is to calculate the hash value according to Eq. (5.4). Then we search each value in the bucket. If we cannot find the entry and the last entry of the linked list is not empty with a valid *offset* value (not zero), we traverse the linked list to find that value. If we want to delete an entry right in the bucket with the same hash value (not the last one), we delete it. If it is the last entry of the bucket, we check if it has a valid *offset* value. If not, we delete it. Otherwise, we use the next element in the linked list to replace the current bucket's last entry. We show this approach using Fig. 5.7.

With the help of the voxel hashing, we can maintain the memory for voxels efficiently. In this chapter, we discussed the primary branch for the 3D reconstruction: Srufel and TSDF. We use the TSDF to obtain the environment 3D model because it is easy to classify the dynamic objects naturally. In the next chapter, we'll discuss the implementation and test of our customized system for a dynamic environment.

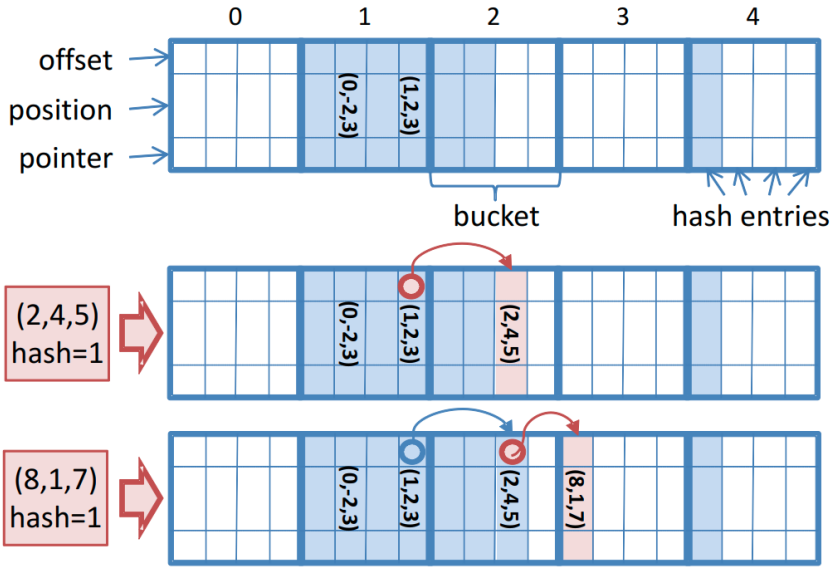


Figure 5.6: In this figure, we insert two hash entries with the same hash value. When we insert (2,4,5), the bucket that has hash value 1 is full. Thus, we store the (2,4,5) in the next bucket's first available position. We set the *offset* value of the (1,2,3) and make it points to the (2,4,5). Then if we want to insert (8,1,7) with hash value 1, we find (1,2,3) and then (2,4,5). We avoid putting the entry with a different hash value than the bucket into the last element, which is reserved for the linked list head. Finally, we put the (8,1,7) into the bucket with a hash value of 3.

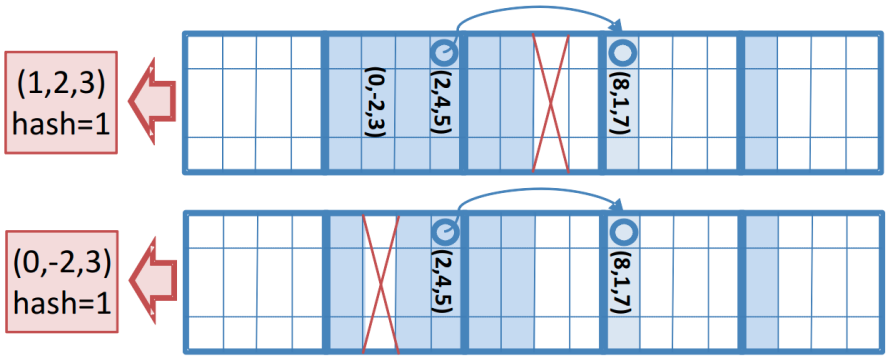


Figure 5.7: In this figure, we delete two hash entries with the same hash value. If we want to delete (0,-2,3) with hash value 1 and it is right in the bucket with hash value 1, we delete it. Suppose we want to delete the (1,2,3) (Fig. 5.6), which is the last entry of the bucket and connects another element in the linked list. We find which one it connects to using the *offset*. The entry connects to (2,4,5); we use it to replace the (1,2,3) and delete the entry where the original (2,4,5) is. Finally, we need to connect (2,4,5) with another entry (8,1,7) already in the linked list.

Chapter 6

Experiments

In this chapter, we discuss the current solutions for 3D reconstruction in the dynamic environment. We present our solution on the CPU and GPU, combing the systems we discussed in the previous chapters.

6.1 Current Solutions

In this section, we will briefaly talk about two systems designed for dense reconstruction in a dynamic environment. One is for indoor scenario – Co-Fusion [96] and the other is for outdoor scenario – DynSLAM [9]. There aren't many systems that can do 3D reconstruction from visual information considering the dynamic objects, so we pick these two representatives. Both of them are built on a 3D reconstruction system for a static environment. The Co-Fusion is built on Elastic Fusion [118], a precise dense reconstruction system which has been shown in the previous chapter, while the other is built on Infinitam [83]. Infinitam is designed for large-scale environment 3D reconstruction. It is interesting to note that Elastic Fusion is a surfel-based system, and Inifinitam is TSDF based system (although the surfel version of Infinitam is under development). Those two systems represent the current high-end dense reconstruction system of two main approaches – surfel and TSDF. Both of them show their abilities for future development in a dynamic environment.

Co-Fusion tries to reconstruct the (potential) dynamic objects and the static background independently. We can choose two methods to segment the dynamic objects. One is geometry-based motion segmentation, which can run in real-time on the CPU, and the other uses Mask R-

CNN-based learning network for object detection, so a strong GPU is needed. Motion segmentation can classify any objects while the other can only classify certain classes from the COCO dataset [66]. Both camera and dynamic object tracking are based on minimizing point-to-plane ICP and photometric errors by comparing two frames' corresponding pixel's RGB intensity. The goal is to find the transformation matrix between frames or models. In equation 6.1, λ is a weight parameter for the RGB error.

$$E_{track} = E_{icp} + \lambda E_{rgb} \tag{6.1}$$

There are several limitations for the error terms in equation 6.1. The ICP error can only be applied to the rigid body [12], and photometric error is applied under the "illumination consistency" assumption [82] because we directly compare the pixel intensity between frames. Figure 6.2 [82] shows the illumination change scenario that is not suitable for photometric based error tracking. What's more, the above system only uses visual information for tracking. The visual-inertial system is proved to have a more robust sensor pose estimation, especially when the sensor movement is rapid [19, 65, 31, 85].

Even ordinary dynamic objects may not be a rigid body such as a person. Figure 6.1 shows an example of deformation that standard ICP cannot handle. We hope our system can take 3D reconstruction for a non-rigid body. From the illumination perspective, we prefer a feature-based tracking so that the sensor tracking could be more robust to the illumination change [75].

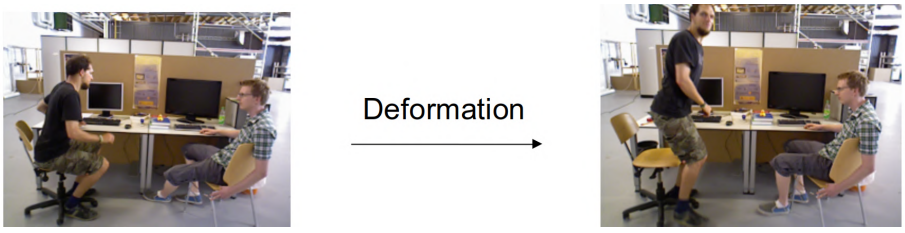


Figure 6.1: Deformation happens when the left person stands up from sitting. ICP cannot handle such a scenario.

DynSLAM has some common features with Co-Fusion. DynSLAM also tries to model the

Figure 6.2: Illumination change scenario generally is not suitable for photometric error based tracking.

dynamic objects and the static background separately. Benefit from the Infinitam, it can do the 3D reconstruction on a large scale, which shows in figure 6.3. It also uses ICP and photometric errors for tracking, which shares the same shortcomings we mentioned before. DynSLAM uses Multi-task Network Cascades [30] (MNC) for segmentation. MNC is a deep neural network architecture, for instance, segmentation like Mask R-CNN. DynSLAM also tries to do 2D tracking besides the 3D reconstruction tracking. The 2D tracking idea is from the classic tracking algorithm SORT [119]. As we have described in the first chapter about object tracking, the 2D matching algorithm tries to match the different objects' track by their bounding box IOU for each segmentation in two frames, which can be further improved by considering each object's feature.

My proposed system has the following strength comparing with the current system:

- We don't try to match the 3D object model independently or use ICP related metrics. Instead, we use some natural properties of TSDF for 3D reconstruction, suitable for a non-rigid body.
- We separate the sensor tracking system and the 3D reconstruction system. The 3D reconstruction needs a correct sensor to pose estimation for fusing points to voxel. We can use the strength of the visual-inertial system to provide a robust pose estimation.
- To enhance the 2D and 3D object tracking, we consider the object features and try to match them.
- We design our proposed method for both indoor and outdoor environments. To adjust to a different scenario, we need to modify the voxel size parameter.

6.2 Proposed Solution

In this section, we are going to propose two kinds of reconstruction systems. One is CPU-based. We use Voxblox as the core reconstruction system; the other is GPU-based. We use the InfiniTAM as the core reconstruction system. However, we need a GPU for image segmentation for

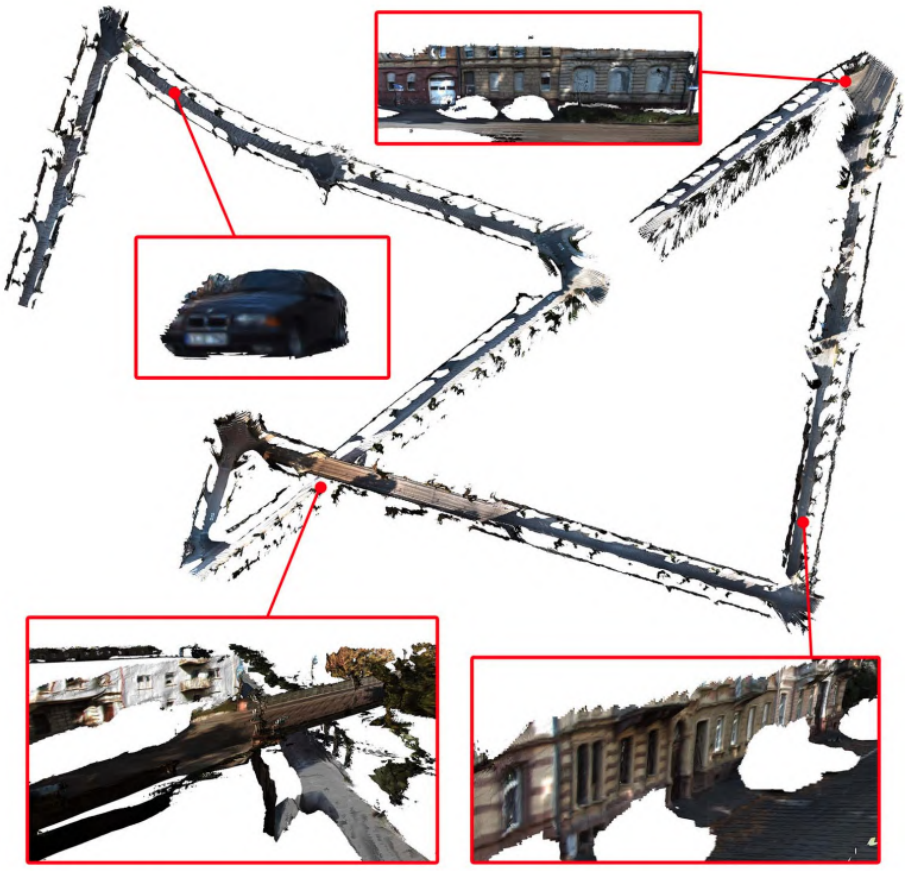


Figure 6.3: DynSLAM reconstruct the 3D object like the car and static background.

both of them. We apply the segmentation offline for the CPU-based approach, while GPU-based, we use the segmentation online.

6.2.1 CPU Based Reconstruction

We draw our proposed system in the figure 6.4. From the perspective of sensor input, We require two types of sensors: camera and IMU for the visual-inertial SLAM pose estimation. We also need three types of images: grayscale stereo image and IMU are sent into the visual-inertial system, RGB and depth image are sent into the Voxblox to generate an RGB point cloud. Realsense series cameras such as D435 and D455 can provide a stereo image pair and RGB-D image pair. For the sensor tracking part, we can see our stereo image is firstly sent into a segmentation system such as Mask R-CNN and Yolact for image segmentation. Then we remove the potential dynamic objects from the image and dispatch the rest to the sparse visual-inertial system. In this way, we'll force the feature detector to detect features outside of the dynamic objects, seen in figure 6.5.

The mask boundary may have a lot of features detected if the feature detector is a heavy gradient-reliable algorithm. This situation is acceptable because if the object is indeed moving, the features on the boundary will be covered in the several subsequent frames and not get matched; if the object is static, they can be matched and used for pose estimation. There is another way to solve this problem: we can still send the entire image to the SLAM system; for the detected features' location in the mask, we remove them and use the rest features. However, like the left side image of figure 6.5, the person who wears a plaid shirt is naturally suitable for feature detection, so many features are attached to that person. If too many features are attached to an object and then removed, we might not have enough features for robust pose estimation. Combining the first and second approaches might be a good test direction in the future. The sensor pose estimation's improvement after removing potential dynamic objects is apparent. We use one of the TUM dynamic RGB-D datasets [108]. The camera has a little translation and rotation in this dataset, but the trajectory estimation shows a significant error without mask removal. After we remove the mask, the trajectory estimation is more close to the groundtruth, shown in figure 6.5.

We sent the segmented image to the visual-inertial SLAM system then the system provided

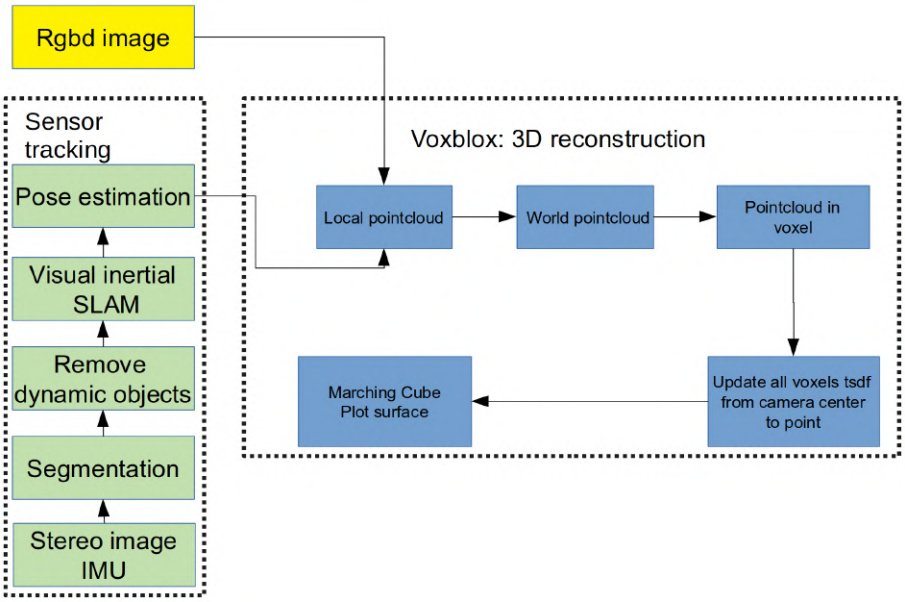


Figure 6.4: Our system is based on a sparse visual-inertial SLAM system for pose estimation (sensor tracking) and Voxblox for dense 3D reconstruction.

an estimated pose to the Voxblox. After receiving the RGBD image, we first convert it to the RGB point cloud by equation 6.2. In equation 6.2, $\mathbf{K}$ is the 3 by 3 camera intrinsics matrix, d is the depth in meters, $\mathbf{p}$ is the 2D pixel in homogeneous coordinate $\mathbf{p} = (u, v, 1)^T$. $\mathbf{P}_c$ is the 3D point. After converting the RGBD image to the camera coordinate 3D points (local point cloud), we convert it to the world coordinate system by multiplying it with the transformation matrix provided by the visual-inertial SLAM, seen in Eq. 6.3. Here we ignore converting the 3D point to the homogeneous form. Now we can decide pointcloud-voxel correspondence by comparing their location. Voxblox uses a hashmap to store all the points in one voxel. The hash key is the voxel index, and the hash value is the point index. Voxblox uses the "grouped raycasting" algorithm to accelerate the TSDF update speed. This algorithm takes the weighted mean of all points and colors within each voxel and performs raycasting only once on this mean position. For a large-scale application, typically, we set a large voxel size too. In this way, one voxel will contain many points, and these points are treated as one point, and the raycasting is done only once. As a result, the

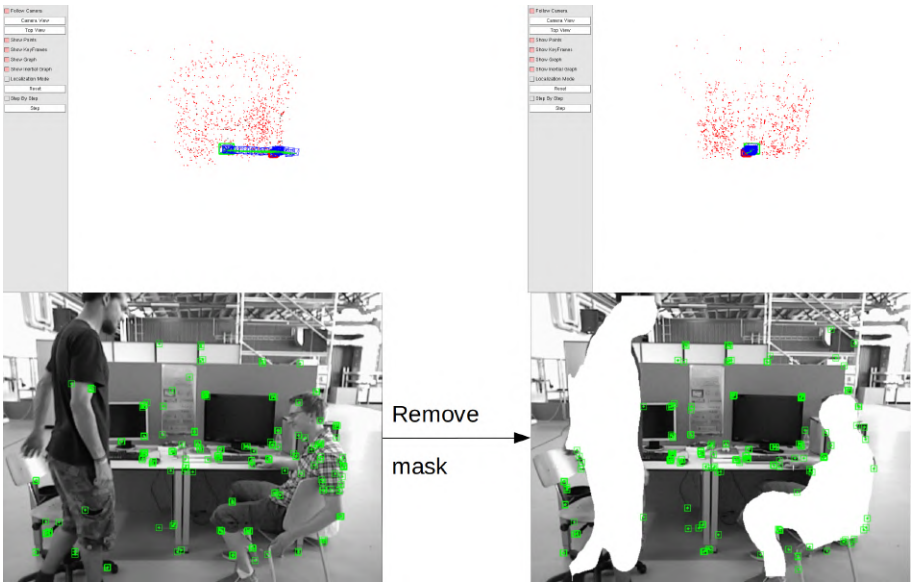


Figure 6.5: The grayscale image in the bottom left is sent to the image segmentation system so that we can detect "persons". Then we remove the potential dynamic objects by setting the corresponding pixel value to a certain number. In this way, We force the feature detector to detect 2D features in other image parts. In the test dataset, the camera is almost static, so the right side's trajectory estimation is closer to the groundtruth.

voxel update procedure accelerates dramatically. After the grouped raycasting, Voxblox updates all the voxels TSDF, weight value from the center to truncation distance δ after the mean point. Finally, we use the marching cube algorithm to render the surface according to the TSDF value.

$$\mathbf{P}_c = d * \mathbf{K}^{-1} * \mathbf{p} \tag{6.2}$$

$$\mathbf{P}_w = \mathbf{T}_w^c * \mathbf{P}_c \tag{6.3}$$

The above description talks about how to deal with the static environment. To enhance the Voxblox to the dynamic environment, we first consider the TSDF value's natural property. TSDF shows the distance between a voxel and its nearest surface. If a new character is inserted or the current surface moves towards the voxel, the absolute distance value will decrease. If a surface is moving away from the voxel, the TSDF value will only increase [40]. From this intuitive explanation, seen from figure 6.6, we can develop an algorithm to decide if the TSDF has a significant change between frames to determine if there are dynamic objects. We want to merge the new TSDF and the original TSDF in a voxel if their difference is smaller than a threshold using equation 5.2. However, if their difference is larger than a threshold, we replace the old one with the new one. This approach is not based on ICP and only detects TSDF value change. We can apply it to the non-rigid body. However, there are shortcomings of this approach that needs to be improved. We discuss them in the result chapter. This algorithm takes the weighted mean of all points and colors within each voxel and performs raycasting only once on this mean position. For a large-scale application, usually, we'll set a large voxel size too. In this way, one voxel will contain many points, and these points are treated as one point, and the raycasting is done only once. As a result, the voxel update procedure accelerates dramatically. After the grouped raycasting, Voxblox updates all the voxels TSDF, weight value from the center to truncation distance δ after the mean point. Finally, we use the marching cube algorithm to render the surface according to the TSDF value.

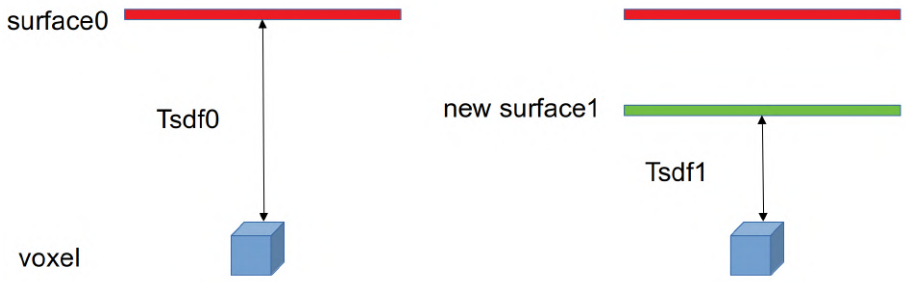


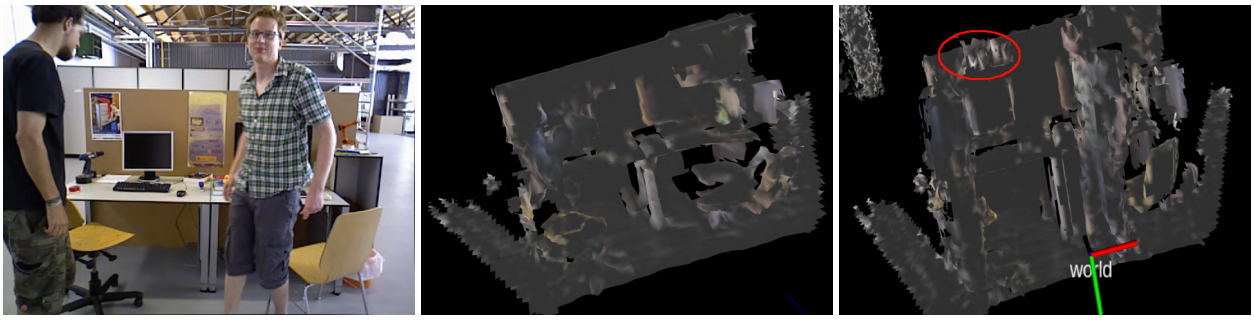
Figure 6.6: Adding an object can only ever decrease the signed distance from each voxel to the next surface, whereas removing an object can only ever increase it [40]. In this image, the cube stands for a voxel, as we can see TSDF1 > TSDF0. If we view the image from the left side to the right side, the TSDF decreases. If we consider the image from the right side to the left side, which means a surface left the scene, the TSDF increases.

6.2.2 Voxblox Result

We implement the system described in the last chapter figure 6.4. The implementation platform CPU is Intel i7 8850H with 2.7G Hz, and the GPU is GTX 1050Ti. We use the GPU for image segmentation. Due to the limitation of the GPU ability, we have to segment the image offline and then reconstruct the 3D scene online. We use ORB-SLAM3 [19] as the visual-inertial system to provide the pose estimation and use Voxblox [80] to do 3D reconstruction. The test dataset is the TUM RGBD dataset with dynamic objects. We chose two of them – "desk with person" and "walking static" to demonstrate the current result. The result is shown in figure 6.7 and 6.8. Both show the results of 3D reconstruction without and with dynamic detection. With dynamic detection, we can already reconstruct persons in the scene, which is a big improvement.

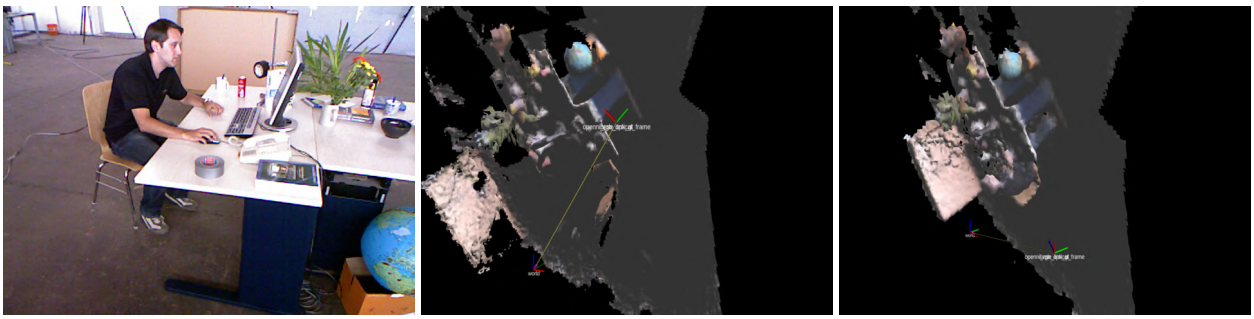
However, there are some shortcomings. First, The "ghost effect" is an interesting research point. It normally happens at the edge of the image. When the dynamic object leaves the current scene, its reconstruction should be destroyed. However, some parts of the reconstruction area are not revisited because of the camera's movement, so those areas' voxels are not revisited, and their TSDF is not updated, becoming "ghost" left there. One possible way to solve this problem is to store the dynamic reconstructed area in a temporary space; then, we revisit the temp area in the next frame. If its back projection to the image is out of the boundary, then we delete that area. We'll discuss another solution in the GPU-based reconstruction. The second shortcoming is speed. Because we use CPU for 3D reconstruction (which is also a strength because this means we can apply this approach to the robot without GPU), the speed is relatively slow. Normally the update frequency is no more than 5 Hz. We developed a GPU version to accelerate the reconstruction speed.

The current demo shows the current path should be correct for dynamic 3D reconstruction. In the video submitted with this proposal, we can also find that in both scenes deformation happens. In figure 6.7, two persons stand up from sitting; In figure 6.8, the person sitting around the desk



(a) Original scene (b) Reconstruction without dynamic detection (c) Reconstruction with dynamic detection

Figure 6.7: The middle image shows the reconstruction without dynamic detection. When the two persons move, the reconstruction is chaotic. The right side of the image shows the reconstruction with dynamic detection. Now we can visualize two persons in the scene. The "ghost effect" happens in the red circle of the right side image because of the camera movement.



(a) Original scene (b) Reconstruction without dynamic detection (c) Reconstruction with dynamic detection

Figure 6.8: It is interesting to see from the middle image that the person is not reconstructed at all even he is right sitting there. If a dynamic object is a little bit far away from the camera and occupy a small part of the image, it may be seen as noise

tries to approach different objects such as a phone, desktop, and books. Those deformations are reconstructed although not delicate. Improve the reconstruction quality is also a future research goal.

We only test the reconstruction on a coarse level in the current application because we cannot process small voxel on the CPU in real-time even with "grouped raycasting". We list the time table we process a frame on Table 6.1. Note we didn't include the segmentation time because it is offline. We obtain the localization time by running different datasets and calculating the average time the SLAM system processes frames. The reconstruction times are an approximation because it is increasing along with the 3D reconstruction size. The voxel size we chose to test is 5cm. The "M1" machine is a laptop with Intel Core i7 8850H and GPU GTX 1050. The "M2" machine is a desktop with an Intel Core i9 10900k, RTX3090 desktop. The total reconstruction is larger than the particular time because we need to synchronize the input between different packages. Our second solution runs the reconstruction on the GPU with the InfiniTAM.

Table 6.1: Table CPU computation time (ms)

	Segmentation	Localization	Reconstruction	Total
M1	/	25	≈ 35	≈ 70
M2	/	20	≈ 25	≈ 50

6.2.3 GPU Based Reconstruction

We work on the InfiniTAM for GPU-based reconstruction. The InfniTAM can run more than 100 Hz on a middle-level GPU and reconstruct an environment with rich details and textures. We don't need to use "grouped raycasting" to accelerate the voxel integration since it is fast enough to process the integration on the GPU. Unlike the Voxblox, the InfiniTAM has an internal tracker to provide pose estimation $\mathbf{T}$ in a static environment. InfiniTAM uses this pose estimation to fuse the point cloud into voxels. To maintain the memory efficiently, the InfiniTAM adopts the voxel hashing we discussed in Section 5.2.1.

InfniTAM's basic voxel structure, voxel block structure, and hash function are the same as the previous work [79]. However, it has some modifications for efficiency. In the InfiniTAM, each bucket contains only one value. If there is a collision, it creates a new bucket to store the new entry. They prove it is more efficient to reduce the overall hash collision, and the coding is simplified because we only need to search one entry in each bucket. The hash entry structure is as follows (same as Section 5.2.1)

```
//Each voxel block maps to a hashentry
struct HashEntry {
  short position[3];
  short offset;
  int pointer;
};
```

We use the *offset* to find the hash entry we want in a linked list if there is a collision. Note we don't need to conduct a linear search like the [79] since each bucket only contains one element. We store the elements that have collisions with the existing hash entry as *unordered entries* while the others as *ordered entries*. This is shown in Fig. 6.9 [54]. The original InfiniTAM has the flow chart shown in Fig. 6.10 (simplified). The InfiniTAM uses various "engines" to process images with depth. It uses the "Image Source Engine" to process various types of images like "Kinect2", "RealSense," "FFMPEG." We convert all input images into a special form. We need to access the image data both on GPU and CPU, so the InfiniTAM creates a customized vector type to store the image data.

```
  template <class T> class Vector4 : public Vector4_ < T >
  {
  public:
    typedef T value_type;
    _CPU_AND_GPU_CODE_ inline int size() const { return 4; }
    _CPU_AND_GPU_CODE_ Vector4() {} // Default constructor
    _CPU_AND_GPU_CODE_ explicit Vector4(const T &t) { this->x = t; this->y = t;
   this->z = t; this->w = t; }
```

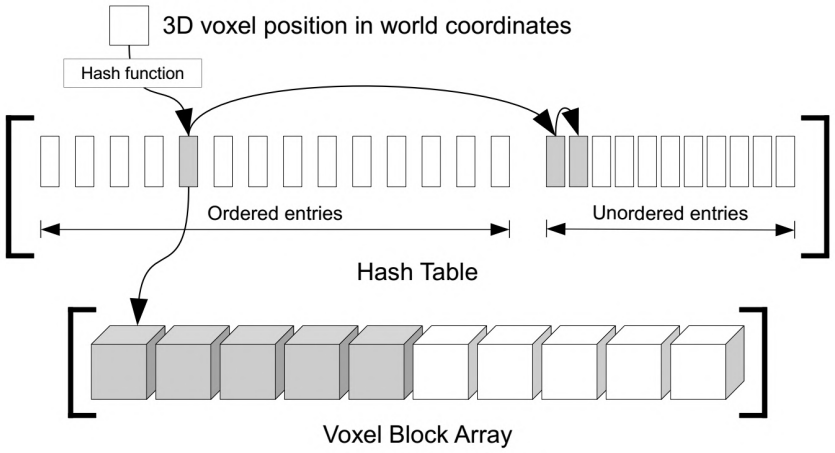


Figure 6.9: Each bucket only has one entry. To retrieve a hash entry, we calculate the hash value and traverse the linked list if the *offset* is valid (not 0). To insert a hash entry, we put the entry with a collision into *unordered entries*. To delete a hash entry, we mark the entry as "removed" instead of deleting it from memory.

```
    ...
}
```

The template type can be *unsigned char* for RGBA image. "_CPU_AND_GPU_CODE_" is either "__device__" (if we use CUDA from GPU programming) or null (if we use CPU only). The InfiniTAM has another "Image" class to store all the vectors as image elements. In this way, we can keep and access pixels stored on the device, and we can also create a customized API to convert the data between the device (GPU) and the host (CPU). After we have the image data in the unique InfiniTAM form, we send the image to the "UI Engine", where we can then process the image data to the "Main Engine." As the name, the main engine is responsible for the core work. Firstly, it updates the UI view according to the image. Secondly, it contains the "track Engine" object to finish the tracking and estimate the pose of the current frame. Thirdly, it sends both the pose estimation and updated view to the "Dense Mapper". The Dense Mapper contains the "Scene Reconstruction Engine" object. It firstly allocates/removes the hash entries according to the new image; then, it integrates the new point cloud from the RGB-D image to the voxels.

```
// allocation
sceneRecoEngine->AllocateSceneFromDepth(scene, view, trackingState, renderState,
    false, resetVisibleList);


// integration
sceneRecoEngine->IntegrateIntoScene(scene, view, trackingState, renderState);
```

Where the *scene* is the 3D world model that contains the voxels, *view* is from the "View Update" block in the main engine. It is responsible for displaying the RGB-D image and filtering. The *trackingState* contains the pose estimation from the tracking engine. The *renderState* includes the render work state for the voxels. The "VisibleList" in the InfiniTAM is used to maintain the voxel blocks that are currently visible. We only need to update the voxel blocks that are currently visible. Thus, the computation can be cheaper.

The original InfiniTAM flow chart is for a static environment. The tracking engine uses

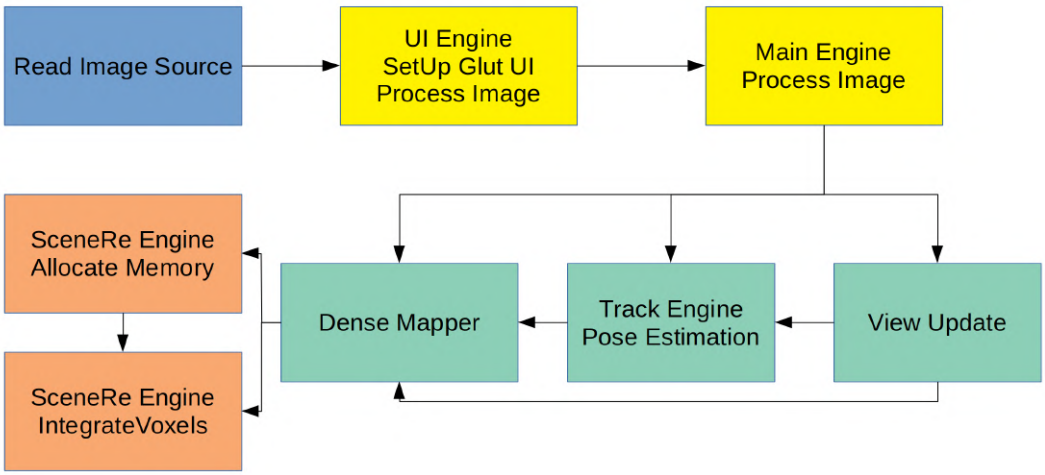


Figure 6.10: The InfiniTAM uses various "engines" to process images. The image source engines read the image and convert it to a particular form. Then we send the image to the UI Engine. UI engine defines the Glut [58] API for user interaction. After that, we send the image to the "Main Engine", which is responsible for view update, pose estimation, and map update. In the Dense Mapper, we allocate memories using voxel hashing and then integrate voxels.

point-to-plane ICP to match the point cloud from one frame to another, requiring a static world. However, in our implementation, we consider not only dynamic objects but also deformation. The tracking system will fail under these circumstances. With an ill pose estimation, the process of integrating the point cloud into voxels will fail. To illustrate this, we use a depth camera to record a video and process this video with InfiniTAM. We use the Fig. 6.11 to demonstrate this process. In Fig. 6.11, There are six blocks in this figure. Each block is a screenshot from the InfiniTAM UI. In each block, there are three small regions. On the right side of the block are the depth image and grayscale image. The left side is 3D reconstruction. In this scenario, the camera is static. The video starts from the top left corner. The reconstruction is delicate initially. However, as the person moves around, the tracker fails to track the camera, so the reconstruction fails. When the person starts rotating at the bottom left side, the reconstruction cannot follow the person. On the right side, the person lifts his hands and stands up, the reconstruction collapse.

To allow the InfiniTAM to work in a dynamic environment, we first need to solve the tracker problem. It is non-trivial to work on the internal tracker because it only relies on the point cloud and point-to-plane ICP. Inspired by our CPU-based reconstruction work, we want to connect the InfiniTAM with an external tracker like the ORBSLAM3. We can use the ORBSLAM3 to provide a robust pose with the segmented images like the CPU-based work. We update our system flow chart like the Fig. 6.12. In Fig. 6.12, we have two input sources. One is the IMU, providing the linear acceleration and rotational velocities. The other is the camera, providing a pair of stereo grayscale, RGB, and depth images. We have three main systems to process the input data. The image segmentation system segments the image and extracts the potential dynamic objects like people and cars. It sends the segmented mask to the visual-inertial system and the InfiniTAM. The visual-inertial system accepts the segmented mask as well as the stereo image, outputs the pose estimation. It sends the pose estimation to the InfiniTAM for voxel integration. The InfiniTAM accepts the segmented mask, RGB-D image, and pose estimation for 3D reconstruction.

However, even with more reliable pose estimation, we still cannot guarantee a precise reconstruction because we need to justify if the points belong to the dynamic objects. With the

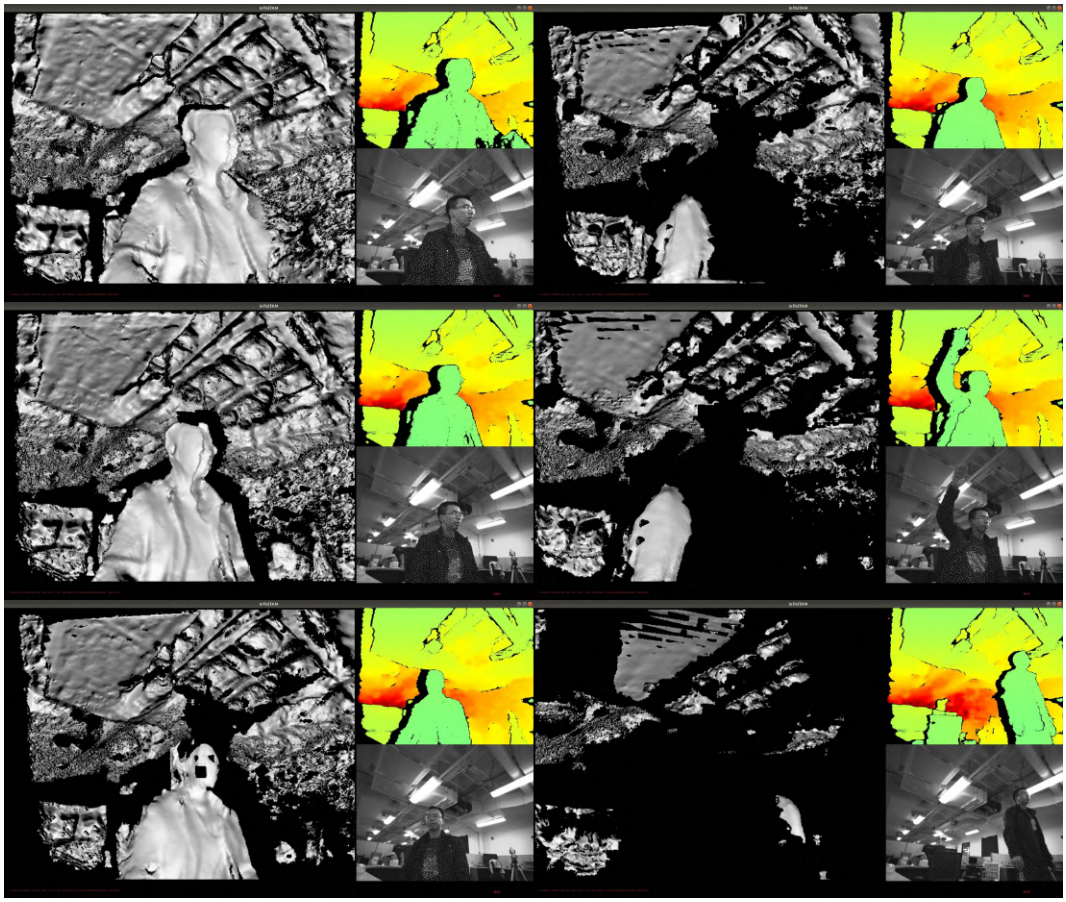


Figure 6.11: There are six blocks in this figure. Each block is a screenshot from the InfiniTAM UI. In each block, there are three small regions. The right side of the block is the depth image and grayscale image. The left side is 3d reconstruction. The video starts from the top left corner. As the person starts moving around, the reconstruction fails.

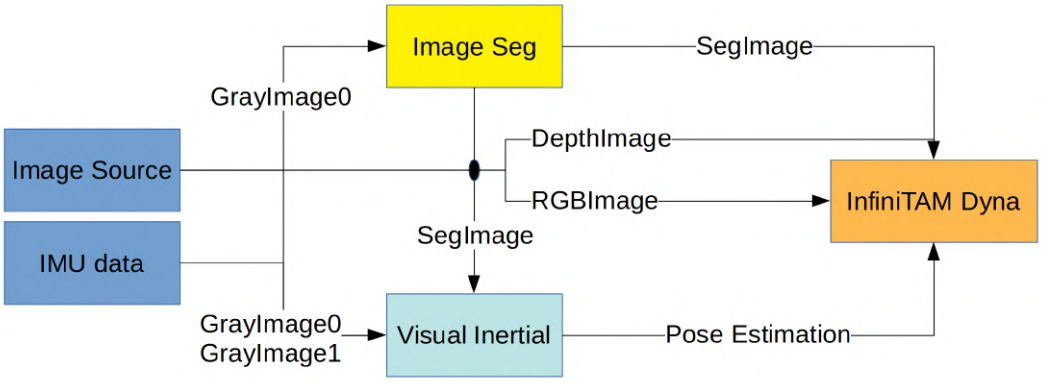


Figure 6.12: We have two input sources: The IMU data has linear acceleration and rotational velocities. The images include RGB-D as well as grayscale stereo image pairs. We have three central systems: the image segmentation system generates the potential dynamic objects mask. The visual-inertial system can provide the pose estimation. The InfiniTAM takes the RGB-D image, mask, and pose estimation as input to reconstruct the 3D environment.

help of the theory shown Fig. 6.6, we can decide if the voxel contains dynamic points according to the tsdf value change. The following pseudo-code illustrates the cheap process.

```
oldF = voxel.sdf;
oldW = voxel.w;
if(abs(newF - oldF)<th){
  newF = oldW * oldF + newW * newF;
  newW = oldW + newW;
  newF /= newW;
  newW = MIN(newW, maxW);
}
voxel.sdf = newF;
voxel.w = newW;
```

The *newF* is from the Eq. (5.2) d_{i+1}. *newW* is 1. If the difference between the *newF* and the *oldF* is smaller than a threshold (we use 0.1m), we assume there are no dynamic objects in this voxel. Then We update the *tsdf* using the typical way in Eq. (5.2). Otherwise, we think there are dynamic objects; then we use the *newF* to replace the *oldF*. This cheap approach can give a more substantial reconstruction than the original InfiniTAM. However, this may lead to a "background float" phenomenon. Due to the noise of the measurement and the customized threshold, the algorithm may also think of the static background as dynamic objects and continue updating the *tsdf* value. This problem leads to a noisy environment; even it is static. We use the mask from the image segmentation system to solve this problem. We only update the voxels that have potential dynamic objects (people, cars, etc.). We show the comparison using Fig. 6.13. However, as we train the model for the dynamic object on limited classes, it indeed happens that we ignore some other dynamic objects. To solve this, we plan to use two thresholds. A lenient threshold *th0* (0.1m) and a strict threshold *th1* (0.3 0.5m). For the voxels in the mask, we use the *th0* to identify the dynamic objects; otherwise, we use the *th1* to identify the dynamic objects.

Considering we have new inputs (image mask and the external pose estimation), we modify the InfiniTAM, and its new flow chart is shown in Fig. 6.14. To make the InifiniTAM be able to

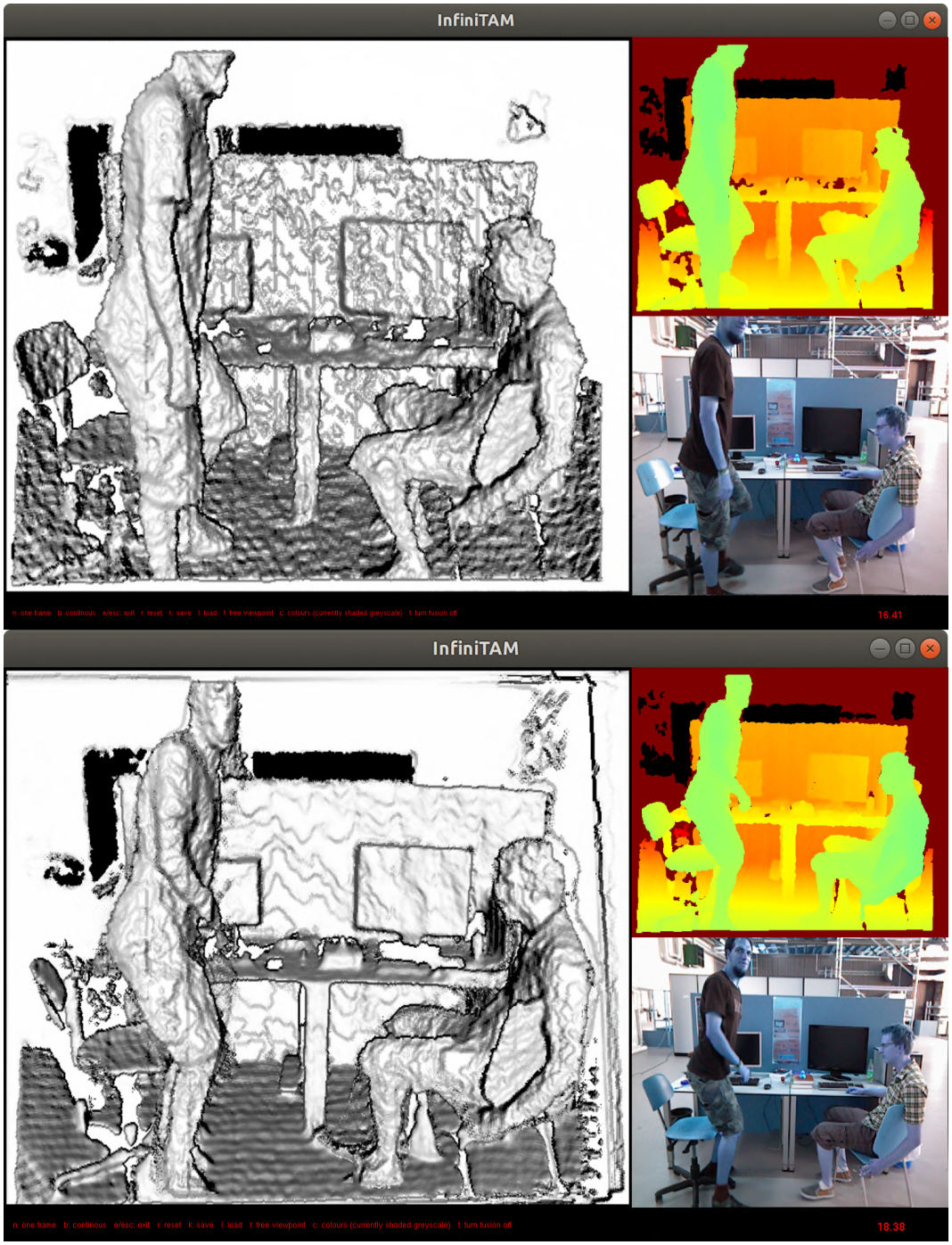


Figure 6.13: In the top figure, we show the reconstruction result simply by using a threshold. The background is noisy because the algorithm measurement is noisy. In the bottom figure, we use a mask from the image segmentation, and we only update the voxels in the mask. Thus, the static background has a solid and more precise reconstruction.

talk to other systems, we use ROS for message communication. We write ROS wrapper for the image segmentation system, visual-inertial system, and the InfniTAM dynamic version. For the InfiniTAM, we first add a "ROS Image Source Engine " to subscribe to the images from the camera and the segmentation system, then we add "External Tracker Engine" to subscribe to the external pose estimation.

consdering people might use InifniTAM for pure static environment reconstruction, we

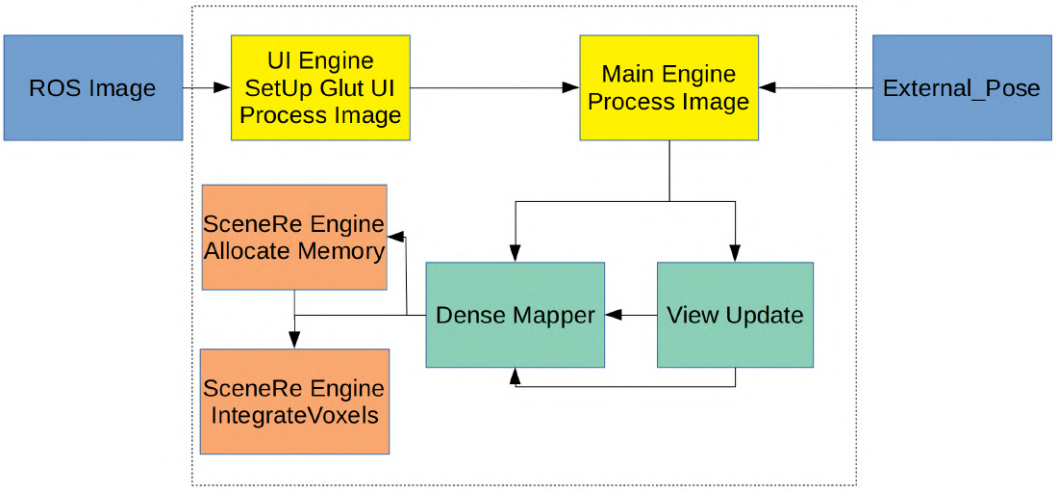


Figure 6.14: We remove the internal tracker of the InfniTAM and use the external tracker from the visual-inertial system. We use the ROS image to unify the input image sources. In the integration part, we consider the dynamic objects according to the mask and the *tsdf* change. The block in the dashed line is the "InfiniTAM Dyna" block in the Fig. 6.12

release the InfiniTAM ros wrapper version independently [1] . This version has the following three features as we talked

- Accpet receiving ROS image topic
- Accpet External pose estimation
- RGB reconstruction

The official InfiniTAM v3 has bugs on RGB reconstruction, and we fix that issue in this package. To allow the InfinTAM to accept the external pose, we need to treat the external tracker as an

[1] `https://github.com/zhaozhongch/infinitam_ros`

input source, creating a "ExternalTrackerEngine.cpp". In this file, a pose subscriber subscribes to the pose messages and pushes them into a queue.

```
void ExternalTrackerEngine::ExternalPoseCallback(const geometry_msgs::PoseStamped
    ::ConstPtr& msg){
    std::lock_guard<std::mutex> guard(tracker_mutex_);
    pose_queue_.push(*msg);
}
```

We also put the incoming image into a queue. When an input image calls the pose estimation, we match the image pose queue by timestamp. It is easy to synchronize the input pose and the grayscale image. However, the mask from the image segmentation system typically has a delay. We use the Yolact for the image segmentation, and it typically needs more than 30 ms to identify the objects in a middle-size image. We modulize our system so, in the future, we can replace the tracker or the image segmentation system without too much effort.

The GPU-based reconstruction still suffers from the "ghost effect". As shown in Fig. 6.15. We have mentioned that one solution is to store the previously visited blocks and revisit them in

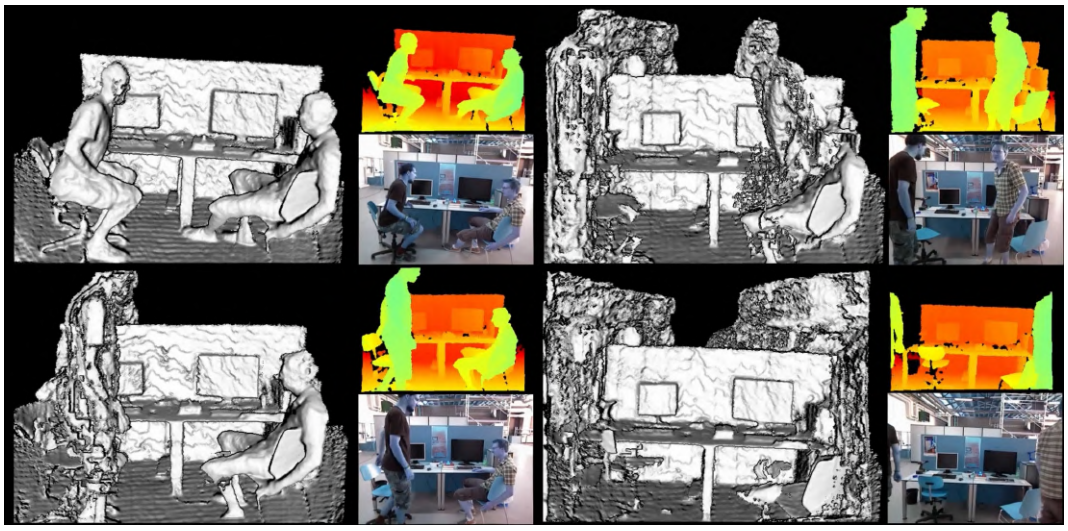


Figure 6.15: As people stand up and walk away in the. People "left" part of their body in the scenario like Fig. 6.7. As the GPU-based reconstruction is more dense, the side effect is more severe.

the next frame. However, as we use a unified model for all the 3D reconstruction, it is non-trivial work. We need to deal with non-visible blocks or deleted blocks in the situation. We propose two other solutions for two different problems.

- We may want to just reconstruct the static environment [100].
- We want to reconstruct both dynamic and static environment.

In the first condition, it is straightforward to mask out the potential dynamic objects. We need to set the *tsdf* value to be none-valid for the voxels that in the mask. We must note we hope to have an "oversized" mask. The mask can never attach to the object without error, as shown in Fig. 3.12. We can tolerate it when the mask covers more space than the object. Once the dynamic objects leave, the voxel can recover the background because the view can naturally see the background and update the voxel-based on the background distance. If the mask cannot cover the object, we'll still face the "ghost effect". A straightforward solution to generate an "oversized" mask is to decide if a voxel neighbor has the voxel in the mask. If so, we can mark that voxel as "in the mask". However, it is expensive to traverse all the neighbors on the device; even it is just a five-by-five patch. We only check if the four corners are in the mask. If so, we mark the voxel in the mask. For different mask qualities, we may need to choose different patch sizes. In Fig. 6.17, we show the result of several distinctive patch size. Although we can relieve the ghost effect by enlarging the patch size, we may not wipe out the ghost effect. If the patch size is too large, we'll face two problems. Firstly, we may miss the mask voxel in the neighborhood. In that situation, we can increase the voxel number to identify moderately. Secondly, the mask is too large to view the background. We need to take care of the trade-off of the patch size.

The mask-out reconstruction is an extra function we provide. This thesis focuses more on reconstructing both dynamic objects and static environments. As discussed before, If the ghost effect occurs, it is a non-trivial work to eliminate it. However, if we compare Fig. 6.13 and Fig. 6.15, we found the former figure doesn't produce a severe ghost effect. This is because in Fig. 6.13

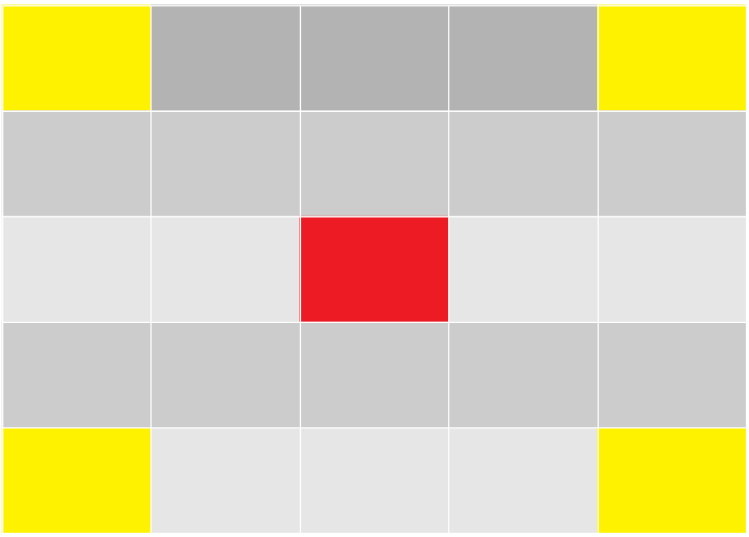
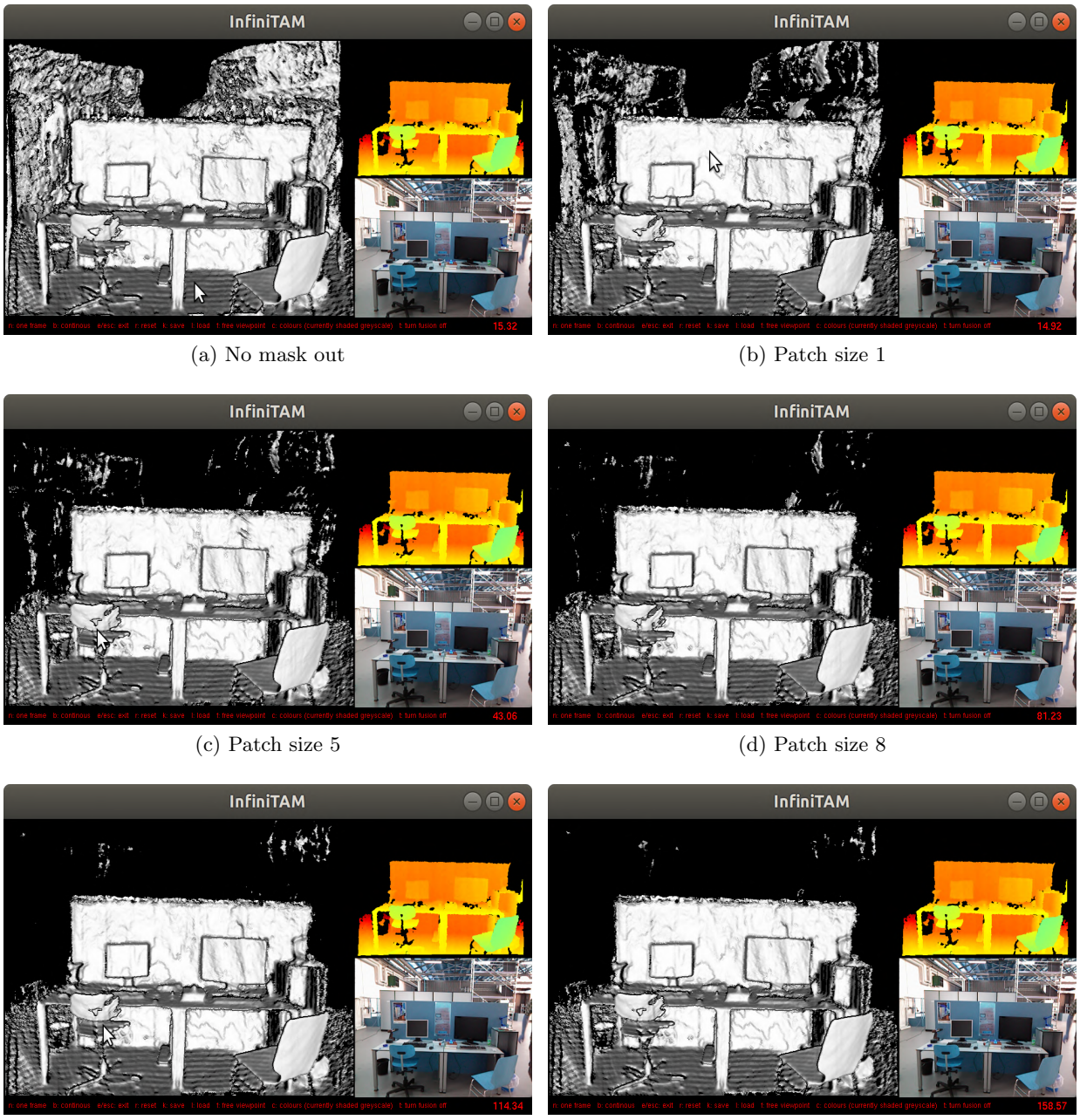


Figure 6.16: It is expensive to traverse all the neighbors of a voxel. We only check if the four corners are in the mask. If so, we mark the voxel in the mask.



(a) No mask out

(b) Patch size 1

(c) Patch size 5

(d) Patch size 8

(e) Patch size 10

(f) Patch size 12

Figure 6.17: We use different "oversized" masks to remove the dynamic objects in the mask and reconstruct the static background. When we enlarge the patch size, we can relieve the ghost effects, but we may not wipe it out.

we use "background compensation" strategy.

We define a maximum distance for reconstruction D_{max} and a minimum distance for re-

construction D_{min}. When the background distance D_b is larger than the D_{max} or invalid, the reconstruction algorithm will not visit that block. Thus, when a dynamic object covers part of the invalid D_b area s_{inv}, it reconstructs the dynamic object. However, when the object leaves, the system will not visit the s_{inv} in the next frame, so the reconstruction doesn't recover to the previous state. If the dynamic object is in front of a valid distance background s_v, the reconstruction naturally recovers to the background after the object leaves. In Fig. 6.15, we notice that when the background has a valid background, where the human body overlaps with the laptop, desk, we don't have a ghost effect even people are moving.

Given this natural phenomenon, we decide to compensate the s_{inv}. In the s_{inv}, there are three situations.

- The depth is larger than D_{max}.
- The depth is smaller than D_{min}.
- The image depth value is invalid due to the camera function.

The TUM-RGBD dataset gives *null* value to the depth of the voxel is too far away. The third situation depends on the camera. For example, in Fig. 6.11, we use the Realsense camera for the depth image. we can notice there is always a black area near the person in the depth image; this is due to the shortcoming of the structured-light camera [98] like the Realsense. To deal with the above situations, we have the following strategies

- For the depth d larger than D_{max}, we let $d = D_{max} - \delta d$. The δd is a small value.
- For the depth d smaller than D_{min}, we do nothing because any objects in front of it won't have reconstruction.
- For the invalid depth, firstly consider hole filling strategy. If not possible, we assign the value $d = D_{max} - \delta d$ and treat it as the background.

We must note if we want to use Internal tracking algorithm in the InfiniTAM to estimate the pose, we cannot mannuly modify the depth. The internal tracking algorithm relies on the ICP, which

requires a resonable depth value. However, we can do that in our system since we rely on external tracking system. In Fig. 6.13, we can see the background is white because we assign its depth using the first strategy. For the condition in Fig. 6.11, we try to fill the empty depth between the human and the background using interpolation. In the Realsense camera, we can use some simple filling methods [49]. The first method is to use the valid neighbors within a radius to fill the hole. The second method is similar. We use left side pixels to compensate the invalid depth since the realsense infer the depth using the left image as the reference. We can choose three ways to compensate the depth. As shown in Fig.6.18.

- Fill the depth with left pixel.
- Fill the depth with the maximum of the left five pixels.
- Fill the depth with the minimum of the left five pixels.

The "persistence filter" also helps—the persistence filter stores the last 7 frames of pixel depth for each pixel. If a pixel has an invalid depth in the current frame, the persistence filter looks up the historical value and uses the latest valid value to fill the current pixel depth. The general hole filling method fills the hole in terms of space, but the persistence filter fills the hole in terms of time. However, if we meet the invalid depth like Fig. 6.11, where a large part of the depth near to the person is invalid, the above naive hole-filling methods may not be able to work. They only consider a small region to fill the invalid depth. We plan to use the persistence filter as well as the mask. If none of the historical depth is valid, i.e., the persistence filter does not work, we use the valid depth at the left side of the pixel to compensate for its depth. Realsense uses the left image as a reference so we can see the invalid depth typically occurs at the left side of the person in Fig. 6.11. The result of our system using the Realsense camera is like Fig. 6.19. For the TUM-RGBD data, we simply replace the invalid depth with $D_{max} - \delta d$. The result is like Fig. 6.13.

Finally, we list the time that each module consumes on Table 6.2 after we run different datasets on our local machine. The voxel size we choose to test is 5mm (10 times smaller than the CPU reconstruction). The "M1" machine is a laptop with Intel Core i7 8850H and GPU GTX

Table 6.2: Table GPU computation time (ms)

	Segmentation	Localization	Reconstruction	Total
M1	102	30	≈ 20	≈ 320
M2	18	22	≈ 4	≈ 48

1050. The "M2" machine is a desktop with an Intel Core i9 10900k, RTX3090 desktop. The total reconstruction is larger than the individual time because we need to synchronize the input between different packages.

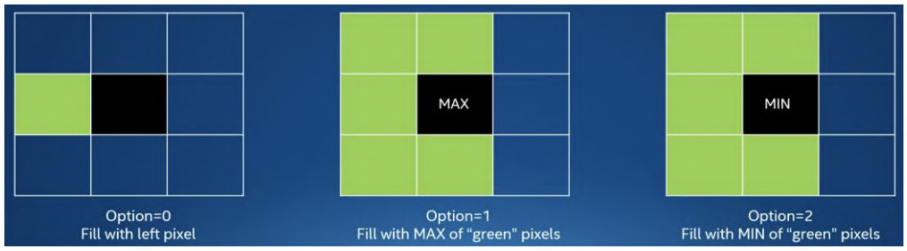
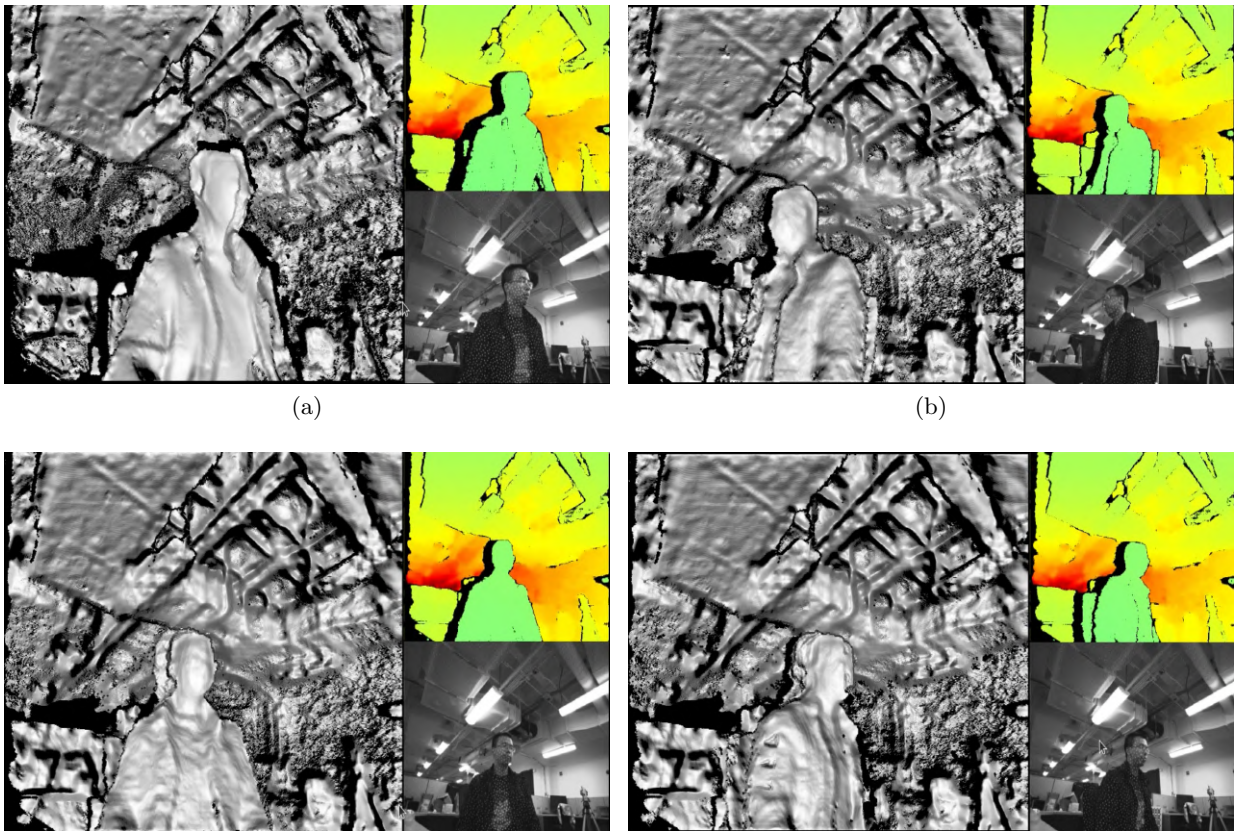


Figure 6.18: Intel Realsense depth fills methods. We can supply the invalid depth using the left pixel, maximum of the left pixels, or a minimum of the left five pixels.

In this chapter, we talk about CPU-based on GPU-based 3D reconstruction. We consider using the mask to detect the potential dynamic objects. We use the *tsdf* value change to identify further if the object is moving. Considering the internal tracker of the InfiniTAM is not robust to the dynamic scene, we connect it to an external tracker. The external tracker can utilize the mask to exclude the potential dynamic objects and then estimate robust poses. We can follow and reconstruct the dynamic scene while the original InfiniTAM or Voxblox cannot do that. This process is generally expensive because of mask detection. Fortunately, we modulize each block in Fig. 6.12. In the future, we can choose a more robust mask detection system or external tracking system as long as they output the same ROS topic. We solve the "ghost effect" by assigning depth value to the invalid pixels. Although this strategy is still under testing, we have already shown its advanced result using the TUM-RGBD dataset. We would consummate the depth compensation



(a) (b)

(c) (d)

Figure 6.19: The result after considering dynamic objects using a realsense camera. Compare to Fig. 6.11, we can follow the person's rotation on the chair.

strategy in the future, and we believe it is a direction to eliminate the ghost effect.

Chapter 7

Conclusion and Future work

In this paper, we show a possible solution to do 3D reconstruction considering the dynamic objects and deformation. The idea behind the dynamic reconstruction is simply considering the TSDF value change. If a voxel has a large TSDF value variance, it is high possible there are dynamic objects "passing through" that voxel. We use an external visual-inertial SLAM system to provide robust pose estimation for the 3D reconstruction system, i.e., we decouple the reconstruction system and the localization system. The localization system can give a robust pose estimation given the segmented image with potential dynamic objects. We modulize our three main components: Reconstruction, segmentation and sensor tracking. We connect the sub-systems through ROS. We can replace each module easily as long as they can share the same type of message with other systems. We apply our dynamic detection method to Voxblox and InfniTAM, two 3d reconstruction libraries for the static environment, and have an remarkable dynamic object reconstruction result. Besides the main reconsruction system, we also share two tutorial systems for understanding the sensor and object segmentations. Currently, there are still several aspects that need to be improved:

- The quality of dynamic 3D objects needs to be improved. If an object is too close to the camera, some area may not have depth information. This leads to more unexpected free space in the 3D reconstruction. we need to compensate the areas that have no depth information. If a pixel has no depth information, we can simply use the left/right side's pixel to compensate its pixel density. However, if a large patch has no depth information, it would be tricky to compensate the depth. It is possible to use learning-based method to

obtain the whole depth information [47], [4] or some other literatures that can compensate part of the depth in the image.

- Our current solution to resolve the "ghost effect" is to add the virtual background. This method can keep the voxel in an update statue. For simplification, we just set the faraway background depth value as the maximal distance our system can reconstruct. This leads to a noisy background reconstruction since they have different depth in reality. In the future, we need to consider an elegant solution to include the background into the reconstruction system.

- The current object detection system mainly works on the "people" tracking because the current evaluation dataset is mainly for people, we plan to retrain and modify their model for at least the following objects: people, cars, trucks.

- The segmentation system is computational expensive. We choose instance segmentation because we thought we might want to track different objects in a future system. However, in many applications, we might just want to reconstruct the dynamic objects regardless of its individual ids. In this case, we can replace the segmentation module by faster semantic segmentation systems [39], [121], [123].

- Find an efficient way to evaluate the dense reconstruction considering the dynamic objects. It is a non-trivial work to assess the dense reconstruction quantitatively. We can use 3D groundtruth equipment (such as Leica 3D scanner) to evaluate our method. Another possible solution is to project the reconstruction to the image and compare the 3D point with the pixel.

- Implement object 3D tracking algorithm and combine our previous filter tuning work [22, 23].

By solving the above problems, we should have a complete 3D reconstruction system for a dynamic and static environment, as well as indoor and outdoor environments. The above research problems

can also construct a comprehensive thesis that reaches different areas of computer vision. The dense reconstruction in a dynamic environment is a non-trival work since it is heavily cross disciplinary. However, we believe with our dynamic object detection stragety and the state of the art tracking and reconstruction algorithms we can achieve the goal.

Bibliography

[1] Sameer Agarwal and Keir Mierle. Ceres Solver: Tutorial & Reference. Google Inc.

[2] Belal Ahmed, T Aaron Gulliver, and Saif alZahir. Image splicing detection using mask-rcnn. Signal, Image and Video Processing, 14(5):1035–1042, 2020.

[3] Daniel Alazard. Introduction to kalman filtering. SUPAERO, 2005.

[4] Amir Atapour-Abarghouei and Toby P Breckon. Real-time monocular depth estimation using synthetic data with domain adaptation via image style transfer. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 2800–2810, 2018.

[5] Tong Bai, Yu Pang, Junchao Wang, Kaining Han, Jiasai Luo, Huiqian Wang, Jinzhao Lin, Jun Wu, and Hui Zhang. An optimized faster r-cnn method based on drnet and roi align for building detection in remote sensing images. Remote Sensing, 12(5):762, 2020.

[6] Tim Bailey and Hugh Durrant-Whyte. Simultaneous localization and mapping (slam): Part ii. IEEE robotics & automation magazine, 13(3):108–117, 2006.

[7] Y Bar-Shalom, X Li, and T.Kirubarajan. Estimation with Applications to Navigation and Tracking. Wiley, New York, 2001.

[8] Ioan Andrei Bârsan, Peidong Liu, Marc Pollefeys, and Andreas Geiger. Robust dense mapping for large-scale dynamic environments. In International Conference on Robotics and Automation (ICRA), 2018.

[9] Ioan Andrei Bârsan, Peidong Liu, Marc Pollefeys, and Andreas Geiger. Robust dense mapping for large-scale dynamic environments. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 7510–7517. IEEE, 2018.

[10] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). Computer vision and image understanding, 110(3):346–359, 2008.

[11] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In European conference on computer vision, pages 404–417. Springer, 2006.

[12] Paul J Besl and Neil D McKay. Method for registration of 3-d shapes. In Sensor fusion IV: control paradigms and data structures, volume 1611, pages 586–606. International Society for Optics and Photonics, 1992.

[13] Alex Bewley, Zongyuan Ge, Lionel Ott, Fabio Ramos, and Ben Upcroft. Simple online and realtime tracking. In 2016 IEEE International Conference on Image Processing (ICIP), pages 3464–3468. IEEE, 2016.

[14] Christopher M. Bishop. Pattern recognition and machine learning. Springer, New York, 2006.

[15] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. arXiv preprint arXiv:2004.10934, 2020.

[16] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact++: Better real-time instance segmentation. arXiv preprint arXiv:1912.06218, 2019.

[17] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In Proceedings of the IEEE international conference on computer vision, pages 9157–9166, 2019.

[18] Eric Brochu, Vlad M Cora, and Nando L B BOTutorial De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. arXiv preprint arXiv:1012.2599, 2010.

[19] Carlos Campos, Richard Elvira, Juan J Gómez Rodríguez, José MM Montiel, and Juan D Tardós. Orb-slam3: An accurate open-source library for visual, visual-inertial and multi-map slam. arXiv preprint arXiv:2007.11898, 2020.

[20] Jiawen Chen, Dennis Bautembach, and Shahram Izadi. Scalable real-time volumetric surface reconstruction. ACM Transactions on Graphics (ToG), 32(4):1–16, 2013.

[21] Yu Chen, Yisong Chen, and Guoping Wang. Bundle adjustment revisited. arXiv preprint arXiv:1912.03858, 2019.

[22] Zhaozhong Chen, Christoffer Heckman, Simon Julier, and Nisar Ahmed. Weak in the nees?: Auto-tuning kalman filters with bayesian optimization. In 2018 21st International Conference on Information Fusion (FUSION), pages 1072–1079. IEEE, 2018.

[23] Zhaozhong Chen, Christoffer Heckman, Simon Julier, and Nisar Ahmed. Time dependence in kalman filter tuning. arXiv preprint arXiv:2108.10712, 2021.

[24] Junhao Cheng, Zhi Wang, Hongyan Zhou, Li Li, and Jian Yao. Dm-slam: A feature-based slam system for rigid dynamic scenes. ISPRS International Journal of Geo-Information, 9(4):202, 2020.

[25] Evgeni V Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Institute for High Energy Physics, Moscow, Russia, Report CN/95-17, 42, 1995.

[26] Gregory S Chirikjian. Stochastic models, information theory, and Lie groups, volume 2: Analytic methods and modern applications, volume 2. Springer Science & Business Media, 2011.

[27] John L Crassidis. Sigma-point kalman filtering for integrated gps and inertial navigation. IEEE Transactions on Aerospace and Electronic Systems, 42(2):750–756, 2006.

[28] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pages 303–312, 1996.

[29] Lis Custodio, Sinesio Pesco, and Claudio Silva. An extended triangulation to the marching cubes 33 algorithm. Journal of the Brazilian Computer Society, 25(1):1–18, 2019.

[30] Jifeng Dai, Kaiming He, and Jian Sun. Instance-aware semantic segmentation via multi-task network cascades. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3150–3158, 2016.

[31] Jeffrey Delmerico and Davide Scaramuzza. A benchmark comparison of monocular visual-inertial odometry algorithms for flying robots. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 2502–2509. IEEE, 2018.

[32] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In 2009 IEEE conference on computer vision and pattern recognition, pages 248–255. Ieee, 2009.

[33] Konstantinos G Derpanis. The harris corner detector. York University, 2, 2004.

[34] Michael Drass, Hagen Berthold, Michael A Kraus, and Steffen Müller-Braun. Semantic segmentation with deep learning: Detection of cracks at the cut edge of glass. Glass Structures & Engineering, pages 1–17, 2020.

[35] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. IEEE robotics & automation magazine, 13(2):99–110, 2006.

[36] Jakob Engel, Vladlen Koltun, and Daniel Cremers. Direct sparse odometry. IEEE transactions on pattern analysis and machine intelligence, 40(3):611–625, 2017.

[37] Jakob Engel, Thomas Schöps, and Daniel Cremers. Lsd-slam: Large-scale direct monocular slam. In European conference on computer vision, pages 834–849. Springer, 2014.

[38] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. International journal of computer vision, 88(2):303–338, 2010.

[39] Mingyuan Fan, Shenqi Lai, Junshi Huang, Xiaoming Wei, Zhenhua Chai, Junfeng Luo, and Xiaolin Wei. Rethinking bisenet for real-time semantic segmentation. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 9716–9725, 2021.

[40] Marius Fehr, Fadri Furrer, Ivan Dryanovski, Jürgen Sturm, Igor Gilitschenski, Roland Siegwart, and Cesar Cadena. Tsdf-based change detection for consistent long-term dense reconstruction and dynamic object discovery. In 2017 IEEE International Conference on Robotics and automation (ICRA), pages 5237–5244. IEEE, 2017.

[41] Max Ferguson, Ronay Ak, Yung-Tsun Tina Lee, and Kincho H Law. Automatic localization of casting defects with convolutional neural networks. In 2017 IEEE international conference on big data (big data), pages 1726–1735. IEEE, 2017.

[42] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. Supplementary material to: Imu preintegration on manifold for efficient visual-inertial maximum-a-posteriori estimation. Technical report, Georgia Institute of Technology, 2015.

[43] Christian Forster, Luca Carlone, Frank Dellaert, and Davide Scaramuzza. On-manifold preintegration for real-time visual–inertial odometry. IEEE Transactions on Robotics, 33(1):1–21, 2016.

[44] Paul Furgale, Joern Rehder, and Roland Siegwart. Unified temporal and spatial calibration for multi-sensor systems. In 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 1280–1286. IEEE, 2013.

[45] Ross Girshick. Fast r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 1440–1448, 2015.

[46] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 580–587, 2014.

[47] Clément Godard, Oisin Mac Aodha, Michael Firman, and Gabriel J Brostow. Digging into self-supervised monocular depth estimation. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 3828–3838, 2019.

[48] Giorgio Grisetti, Rainer Kümmerle, Cyrill Stachniss, and Wolfram Burgard. A tutorial on graph-based slam. IEEE Intelligent Transportation Systems Magazine, 2(4):31–43, 2010.

[49] Anders Grunnet-Jepsen and Dave Tong. Depth post-processing for intel® realsense™ d400 depth cameras. New Technologies Group, Intel Corporation, 3, 2018.

[50] Shuangquan Han and Zhihong Xi. Dynamic scene semantics slam based on semantic segmentation. IEEE Access, 8:43563–43570, 2020.

[51] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 2961–2969, 2017.

[52] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 770–778, 2016.

[53] Matthew Hoffman, Eric Brochu, and Nando De Freitas. Portfolio Allocation for Bayesian Optimization. Conference on Uncertainty in Artificial Intelligence, pages 327–336, 2011.

[54] Olaf Kähler, Victor Adrian Prisacariu, Carl Yuheng Ren, Xin Sun, Philip Torr, and David Murray. Very high frame rate volumetric integration of depth images on mobile devices. IEEE transactions on visualization and computer graphics, 21(11):1241–1250, 2015.

[55] Rudolph E Kalman and Richard S Bucy. New results in linear filtering and prediction theory. Journal of Basic Engineering, 83(1):95–108, 1961.

[56] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In 2007 6th IEEE and ACM international symposium on mixed and augmented reality, pages 225–234. IEEE, 2007.

[57] Matthew Klingensmith, Ivan Dryanovski, Siddhartha S Srinivasa, and Jizhong Xiao. Chisel: Real time large scale 3d reconstruction onboard a mobile device using spatially hashed signed distance fields. In Robotics: science and systems, volume 4, page 1. Citeseer, 2015.

[58] Joseph Konczal et al. Glut/tk: Open gl with tcl/tk. 2003.

[59] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[60] Harold W Kuhn. The hungarian method for the assignment problem. Naval research logistics quarterly, 2(1-2):83–97, 1955.

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

[62] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11):2278–2324, 1998.

[63] Sangho Lee and Taesoo Kim. Leaking uninitialized secure enclave memory via structure padding. arXiv preprint arXiv:1710.09061, 2017.

[64] Stefan Leutenegger, Margarita Chli, and Roland Y Siegwart. Brisk: Binary robust invariant scalable keypoints. In 2011 International conference on computer vision, pages 2548–2555. Ieee, 2011.

[65] Stefan Leutenegger, Simon Lynen, Michael Bosse, Roland Siegwart, and Paul Furgale. Keyframe-based visual–inertial odometry using nonlinear optimization. The International Journal of Robotics Research, 34(3):314–334, 2015.

[66] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In European conference on computer vision, pages 740–755. Springer, 2014.

[67] Chen Long, Ai Haizhou, Zhuang Zijie, and Shang Chong. Real-time multiple people tracking with deeply learned candidate selection and person re-identification. In ICME, 2018.

[68] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 3431–3440, 2015.

[69] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. ACM siggraph computer graphics, 21(4):163–169, 1987.

[70] David G Lowe. Distinctive image features from scale-invariant keypoints. International journal of computer vision, 60(2):91–110, 2004.

[71] Will Maddern, Geoff Pascoe, Chris Linegar, and Paul Newman. 1 Year, 1000km: The Oxford RobotCar Dataset. The International Journal of Robotics Research (IJRR), 36(1):3–15, 2017.

[72] Will Maddern, Geoffrey Pascoe, Matthew Gadd, Dan Barnes, Brian Yeomans, and Paul Newman. Real-time kinematic ground truth for the oxford robotcar dataset. arXiv preprint arXiv: 2002.10152, 2020.

[73] Ammar Mahmood, Ana Giraldo Ospina, Mohammed Bennamoun, Senjian An, Ferdous Sohel, Farid Boussaid, Renae Hovey, Robert B Fisher, and Gary A Kendrick. Automatic hierarchical classification of kelps using deep residual features. Sensors, 20(2):447, 2020.

[74] Mohinder Grewal and Angus Andrews. Van Loan's Method for Computing Q_k from Continuous Q. In Kalman Filtering: Theory and Practice with MATLAB, pages 150–152. 2015.

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

[76] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In 2011 10th IEEE International Symposium on Mixed and Augmented Reality, pages 127–136. IEEE, 2011.

[77] Timothy S Newman and Hong Yi. A survey of the marching cubes algorithm. Computers & Graphics, 30(5):854–879, 2006.

[78] Gregory M. Nielson. On marching cubes. IEEE Transactions on visualization and computer graphics, 9(3):283–297, 2003.

[79] Matthias Nießner, Michael Zollhöfer, Shahram Izadi, and Marc Stamminger. Real-time 3d reconstruction at scale using voxel hashing. ACM Transactions on Graphics (ToG), 32(6):1–11, 2013.

[80] Helen Oleynikova, Zachary Taylor, Marius Fehr, Roland Siegwart, and Juan Nieto. Voxblox: Incremental 3d euclidean signed distance fields for on-board mav planning. In 2017 Ieee/rsj International Conference on Intelligent Robots and Systems (iros), pages 1366–1373. IEEE, 2017.

[81] Chanoh Park, Soohwan Kim, Peyman Moghadam, Clinton Fookes, and Sridha Sridharan. Probabilistic surfel fusion for dense lidar mapping. In Proceedings of the IEEE International Conference on Computer Vision Workshops, pages 2418–2426, 2017.

[82] Seonwook Park, Thomas Schöps, and Marc Pollefeys. Illumination change robustness in direct visual slam. In 2017 IEEE international conference on robotics and automation (ICRA), pages 4523–4530. IEEE, 2017.

[83] Victor Adrian Prisacariu, Olaf Kähler, Stuart Golodetz, Michael Sapienza, Tommaso Cavallari, Philip HS Torr, and David W Murray. Infinitam v3: A framework for large-scale 3d reconstruction with loop closure. arXiv preprint arXiv:1708.00783, 2017.

[84] Tong Qin, Shaozu Cao, Jie Pan, and Shaojie Shen. A general optimization-based framework for global pose estimation with multiple sensors, 2019.

[85] Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. IEEE Transactions on Robotics, 34(4):1004–1020, 2018.

[86] Carl Edward Rasmussen and Christopher K. I. Williams. Gaussian processes for machine learning. Adaptive computation and machine learning. MIT Press, Cambridge, Mass, 2006.

[87] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 779–788, 2016.

[88] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 7263–7271, 2017.

[89] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[90] Joern Rehder, Janosch Nikolic, Thomas Schneider, Timo Hinzmann, and Roland Siegwart. Extending kalibr: Calibrating the extrinsics of multiple imus and of individual axes. In 2016 IEEE International Conference on Robotics and Automation (ICRA), pages 4304–4311. IEEE, 2016.

[91] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In Advances in neural information processing systems, pages 91–99, 2015.

[92] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention, pages 234–241. Springer, 2015.

[93] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In European conference on computer vision, pages 430–443. Springer, 2006.

[94] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In 2011 International conference on computer vision, pages 2564–2571. Ieee, 2011.

[95] M. Runz, M. Buffier, and L. Agapito. Maskfusion: Real-time recognition, tracking and reconstruction of multiple moving objects. In 2018 IEEE International Symposium on Mixed and Augmented Reality (ISMAR), pages 10–20, 2018.

[96] Martin Rünz and Lourdes Agapito. Co-fusion: Real-time segmentation, tracking and fusion of multiple objects. In 2017 IEEE International Conference on Robotics and Automation (ICRA), pages 4471–4478. IEEE, 2017.

[97] Muhamad Risqi U Saputra, Andrew Markham, and Niki Trigoni. Visual slam and structure from motion in dynamic environments: A survey. ACM Computing Surveys (CSUR), 51(2):1–36, 2018.

[98] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. Kinect range sensing: Structured-light versus time-of-flight kinect. Computer vision and image understanding, 139:1–20, 2015.

[99] Thomas Schops, Torsten Sattler, and Marc Pollefeys. Bad slam: Bundle adjusted direct rgb-d slam. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 134–144, 2019.

[100] Raluca Scona, Mariano Jaimez, Yvan R Petillot, Maurice Fallon, and Daniel Cremers. Staticfusion: Background reconstruction for dense rgb-d slam in dynamic environments. In 2018 IEEE International Conference on Robotics and Automation (ICRA), pages 3849–3856. IEEE, 2018.

[101] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, and Nando de Freitas. Taking the Human Out of the Loop: A Review of Bayesian Optimization. Proceedings of the IEEE, 104, 2016.

[102] G. Shi, X. Xu, and Y. Dai. Sift feature point matching based on improved ransac algorithm. In 2013 5th International Conference on Intelligent Human-Machine Systems and Cybernetics, volume 1, pages 474–477, 2013.

[103] Gabe Sibley, Larry Matthies, and Gaurav Sukhatme. A sliding window filter for incremental slam. In Unifying perspectives in computational and robot vision, pages 103–112. Springer, 2008.

[104] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[105] Joan Sola, Jeremie Deray, and Dinesh Atchuthan. A micro lie theory for state estimation in robotics. arXiv preprint arXiv:1812.01537, 2018.

[106] Hauke Strasdat, José MM Montiel, and Andrew J Davison. Visual slam: why filter? Image and Vision Computing, 30(2):65–77, 2012.

[107] Jörg Stückler and Sven Behnke. Multi-resolution surfel maps for efficient dense 3d modeling and tracking. Journal of Visual Communication and Image Representation, 25(1):137–147, 2014.

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

[109] Johan AK Suykens and Joos Vandewalle. Least squares support vector machine classifiers. Neural processing letters, 9(3):293–300, 1999.

[110] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H Gross. Optimized spatial hashing for collision detection of deformable objects. In Vmv, volume 3, pages 47–54, 2003.

[111] Andrea Vedaldi. An open implementation of the sift detector and descriptor. UCLA CSD, 2007.

[112] Qiang Wang, Li Zhang, Luca Bertinetto, Weiming Hu, and Philip HS Torr. Fast online object tracking and segmentation: A unifying approach. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1328–1338, 2019.

[113] Youbing Wang and Shoudong Huang. Towards dense moving object segmentation based robust dense rgb-d slam in dynamic scenarios. In 2014 13th International Conference on Control Automation Robotics & Vision (ICARCV), pages 1841–1846. IEEE, 2014.

[114] Zhongdao Wang, Liang Zheng, Yixuan Liu, Yali Li, and Shengjin Wang. Towards real-time multi-object tracking. arXiv preprint arXiv:1909.12605, 2019.

[115] Long Wen, Xinyu Li, and Liang Gao. A transfer convolutional neural network for fault diagnosis based on resnet-50. Neural Computing & Applications, 32(10), 2020.

[116] Diana Werner, Ayoub Al-Hamadi, and Philipp Werner. Truncated signed distance function: experiments on voxel size. In International Conference Image Analysis and Recognition, pages 357–364. Springer, 2014.

[117] Thomas Whelan, Michael Kaess, Maurice Fallon, Hordur Johannsson, John J Leonard, and John McDonald. Kintinuous: Spatially extended kinectfusion. 2012.

[118] Thomas Whelan, Stefan Leutenegger, R Salas-Moreno, Ben Glocker, and Andrew Davison. Elasticfusion: Dense slam without a pose graph. Robotics: Science and Systems, 2015.

[119] Nicolai Wojke, Alex Bewley, and Dietrich Paulus. Simple online and realtime tracking with a deep association metric. In 2017 IEEE international conference on image processing (ICIP), pages 3645–3649. IEEE, 2017.

[120] Zhixin Yan, Mao Ye, and Liu Ren. Dense visual slam with probabilistic surfel map. IEEE transactions on visualization and computer graphics, 23(11):2389–2398, 2017.

[121] Changqian Yu, Changxin Gao, Jingbo Wang, Gang Yu, Chunhua Shen, and Nong Sang. Bisenet v2: Bilateral network with guided aggregation for real-time semantic segmentation. International Journal of Computer Vision, pages 1–18, 2021.

[122] Chao Yu, Zuxin Liu, Xin-Jun Liu, Fugui Xie, Yi Yang, Qi Wei, and Qiao Fei. Ds-slam: A semantic visual slam towards dynamic environments. In 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 1168–1174. IEEE, 2018.

[123] Juntang Zhuang, Junlin Yang, Lin Gu, and Nicha Dvornek. Shelfnet for fast semantic segmentation. In Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops, pages 0–0, 2019.