Technical University of Munich

TUM Department of Civil, Geo and Environmental Engineering

Chair of Computational Modeling and Simulation

# Modification of Parameters in Image-based Automated 3D-Reconstruction for Fine Structures

Master Thesis

for the Master of Science Course Civil Engineering

| | |
|---|---|
| Author: | |
| Student ID: | |
| Supervisor: | Prof. Dr.-Ing. André Borrmann |
| | M.Sc. Felix Eickeler |
| Date of Issue: | 20. April 2020 |
| Date of Submission: | 30. November 2020 |

## Abstract

Nowadays with the BIM system advances, digital construction becomes more necessary than ever. While many digital data serve for design and management stages, this thesis focus on monitoring part by generating 3D-model from captured images, which brings benefits to the construction and maintaining stages as well as possibly more general use cases.

Colmap is an open source software which is mainly used in this study. It provides a complete workflow from source images to sparse and dense model, with plenty of parameters to be customized. By following the pipeline of Colmap, fundamental relating theories in Computer Vision such as SIFT, SfM and MVS are explored and utilized.

In this thesis, the analyzation is currently restricted in patch match step of dense modelling. An evaluation system aiming at tower crane is set up for depth-maps to reflect the quality of reconstruction. Additionally, an innovating automatic optimization concept is proposed based on this evaluation.

Through experiments on provided dataset of construction site, the thesis discusses the influence of some critical factors to the reconstruction and verifies the proposed evaluation method at the same time. This study provides an attainment from general reconstruction to targeted construction site and leaves a fundamental system for further research.

# Contents

# List of Abbreviations

| | |
|---|---|
| 2D | Two-dimensional |
| 3D | Three-dimensional |
| BIM | Building Information Modelling |
| CMS | Chair of Computational Modeling and Simulation |
| CV | Computer Vision |
| DSP | Domain-Size Pooling |
| MVS | Multi-View Stereo |
| NCC | Normalized Cross Correlation |
| RGB | Red Green Blue |
| ROI | Region of Interest |
| SfM | (incremental) Structure-from-Motion |
| SIFT | Scale-Invariant Feature Transform |
| UAV | Unmanned Aerial Vehicle |

## Typographical Conventions

The following typographical conventions are used in this thesis:

To denote a code segment or instructions in command-line, *Cambria italic* is used, e.g. *docker run …*, *cv2.threshold(img, 40, 255, cv2.THRESH_BINARY)*.

*Italic* text after hash (#) refers to comment of code, e.g. *# this is a comment*.

Backticks (`) are used for file name and path. General file path in natural language segmented with slash ( / ) is to substitute, e.g. `/path/to/file`, `to/sparse/path`.

# 1 Introduction and Motivation

## 1.1 Image-based Reconstruction

With the popularity of BIM concept, the digital concept permeates through the whole construction field, demands from engineering grows from 2D to 3D even faster. People require an interactive way to keep track of from design-stage to monitoring. While BIM system provides digital workflow following the sequence of building, image-based reconstruction generates the model in an inverse way.

Automated 3D reconstruction from images has been a core computer vision problem for years. [1] For construction sites, it might be just the beginning to make use of these digital mediums. The goal of image-based 3D reconstruction is to infer the 3D geometry and spatial relationship from 2D images. This long standing ill-posed problem is fundamental to many applications such as robot navigation, object recognition, scene understanding, 3D modeling and industrial control etc. [2]

To recovering the lost dimension from just 2D images, we focus on Stereo-based techniques [3] in this thesis. That is, to require matching features across images captured from slightly different viewing angles, and then use the triangulation principle to recover the 3D coordinates of the image pixels. We use Colmap, an open source software to generate both sparse model and dense model. A fused dense model as final result can be visualized in form of point-cloud.

Compared with other methods, image-based modelling methodology is low-cost, flexible and able to produce very accurate models, even of complex objects, comparable with 3D scanner clouds in terms of point density and accuracy. [4]

## 1.2 Applicable Scenarios

Due to the simplicity and flexibility of image capture or even collection from internet, the image-based reconstruction can bring benefits to various industries. The spatial relationship simplifies the 3D building with true ratio and provides depth map to each flat image. Thus, a straight-forward object isolation by extracting a depth range in depth map can be realized, as introduced in this thesis. A final dense model in point-cloud gives incredible possibility of interaction inside a real scene.

Moving Picture Experts Group (MPEG) [6] listed some use cases such as content VR viewing with Interactive Parallax, Geographic Information Systems and Autonomous Navigation Based on Large-Scale 3D Dynamic Maps. As for the construction site, this modelling may bring more possibilities by mapping with a vectorized model of BIM. With a huge set of metadata, it could deliver a much more detailed demonstration for project, or error detection while construction by comparing the real situation with a 3D plan. And this technique may also help with restoration of ancient buildings. For example, it could be possible to restore a full and accurate model of the burned down Notre Dame de Paris with plenty internet images from any camera at any time.

As for capturing, with the mature of the Unmanned Aerial Vehicle (UAV), the cost of a drone reduces significantly, and real-time monitoring thus becomes much easier. Using drone to monitor construction site is a flexible approach with low manpower costs, and it can easily reach the site that is sometimes too hard for human. To keep monitoring and recording on river or mountains, for example, a drone can send back comprehensive information over facade. And this working mode might replace even more traditional manpower with the development of technique and computation. The image-based reconstruction, as one of the various techniques, keeps updating by improving its accuracy and efficiency to help realize a more reliable digital construction.

## 1.3 Aims and Objectives

This thesis explore the mechanism of some classic algorithms in CV, such as Scale-Invariant Feature Transform (SIFT), Structure-from-Motion (SfM) and Multi-View Stereo (MVS). These theories are adopted in the open source software — Colmap. We perform the reconstruction separately in mainly two stages, i.e. sparse model and dense model, as divided by Colmap. By doing experiments with different values for adjustable parameters, influence of some critical parameters are analyzed. In this thesis, we also discuss the effect of DSP-SIFT in feature extraction and how to perform an improvement in patch-match stage for a better dense model.

To evaluate the quality of reconstruction, we use a manually labelled image for benchmark. We assume that the labelled image stands for ground-truth. The analyzation is realized by pixel-wise image comparisons as well as manually comparing in depth-maps and point-clouds. The difference between masks of automated depth map and ground-truth is judged in error of pixels. The objective of the optimization is to minimize the error by ensuring a good precision and recall. Although the final model is not fully optimized, this study provides a relatively comprehensive and meticulous methodology with plenty of experiments and illustrations.

## 1.4 Layout of this Thesis

**Chapter 1** points out the significance why image-based reconstruction is needed based on current situation and trend. Here the feasibility of this technique are shown in described scenarios. And this chapter announces the objective of this study and a brief procedure for evaluation and optimization.

**Chapter 2** introduces the theoretical background over feature detection and image matching of SIFT, as well as the theories over SfM and MVS, which are adopted in the software Colmap.

**Chapter 3** first lists the tools involved in this study and then demonstrate the workflow of reconstruction. Afterwards, a novel method is proposed for evaluating the quality of depth map in order to reflect the quality of reconstruction. Based on this evaluation, a concept of iterative optimization is formulated.

**Chapter 4** shows some specific test cases with regard to the assumptions, including modified configuration on the model, different parameter combinations and their corresponding results. By analyzing distinguishable depth-maps and point-clouds, a better parameter combination is to find, and the evaluation methodology is to test and verify.

**Chapter 5** discusses some further influence on reconstruction apart from Colmap parameters as well as several error-prone occasions, and analyzes the drawback of this study. Some ideas for current method are proposed to achieve a more accurate and stable optimization in the future.

# 2 Theoretical Background

This chapter explores the theoretical basis for studying into the research goal. To lay the foundation, required knowledge in Computer Vision will be introduced. The core technology — Scale-Invariant Feature Transform (SIFT) theory, which is also important in Colmap, brings important terminologies like feature, scale and descriptor. In this way, the corresponding image process and the workflow of SIFT will be interpreted. Later comes the variation of SIFT, Domain Size Pooling (DSP), which provides an efficient improvement on descriptor and matching. In addition to SIFT, further relating theories of Colmap concerning Structure from Motion (SfM) and Multi-View Stereo (MVS) are also introduced. This open source software acts as an application form based on these theories.

## 2.1 Feature Extraction with SIFT

Different from what the naked eye sees, computer or other machines treat a picture by analyzing its mathematical characteristics and grabbing all the features as a result of its comprehension. According to SIFT [7], there are 4 major stages of computation used to generate the image features: scale-space extrema detection, keypoint localization, orientation assignment and keypoint descriptor.

### 2.1.1 Keypoint and Localization

**The Scale Space and Kernel**

Before dealing with features, there are some basic terminologies to figure out. In image processing, scale-space is a technique for representing images at different scales. An image, which originates from a real object, exists as meaningful entities only over certain ranges of scale.[8] In other words, what catch our attention typically depends on how close we look at an object. The forest is something we can expect from the view of a drone. Coming closer, we are then interested in trees or even leaves. Different scales are appropriate for describing different objects in the image.[9]

"Scale-Space Theory" describes a formal theory for representing the notion of scale in image data, which applies to extract features in computer vision.[8] And the kernel is the tool to generate different scale space. It's a small matrix, which can be used for blurring in this case. Doing a convolution between a kernel and an image can produce an image result with blur effect.

**Blurring and Downsampling**

As mentioned by D. G. Lowe. (2004), it has been shown by Koenderink (1984) and Lindeberg (1994) that under a variety of reasonable assumptions the only possible scale-space kernel is the Gaussian function.

Gaussian Blur (Gaussian smoothing) is the way to get rid of tiny details. By doing Gaussian burr on the original image, we could get the blurred-out images.

The scale space of an image is defined as a function L(x, y, σ), that is produced from the convolution of a variable-scale Gaussian, G(x, y, σ), with an input image I(x, y):[8]

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

The scale parameter sigma acts as a regulator for Gaussian blur operator. Increasing the sigma, it increases the blur effect. Some of the details are thus less visible, while some details remain even with stronger blurs. In this way, we got the different scales for the image. While I indicates the original image, L represents the Gaussian smoothed image. Gaussian blur is also used for enlarging the contour of object (stencil) in Chapter 3.3.3 in this thesis.

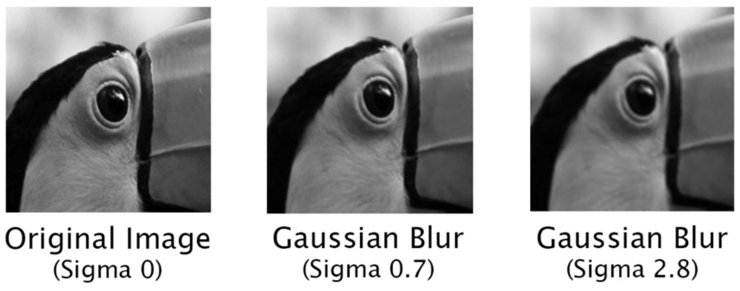


Figure 2.1: A set of images blurred with a Gaussian Blur at different sigma. [10]

A set of blurs for one image with different sigma are included in one octave. In the first octave, a set of progressively blurred-out images based on the original image size are generated. After each octave, the Gaussian image is down-sampled by a factor of 2, and the process repeated. [8] Downsampling should be done over a previously blurred (smoothed) image to avoid aliasing, as shown in Figure 2.2. [9]

The number of octaves and scale depends on the size of the original image. And Lowe suggests that 4 octaves and 5 blur levels are ideal for the algorithm. With the powerful server we increase the number of octaves to 6 in this study.

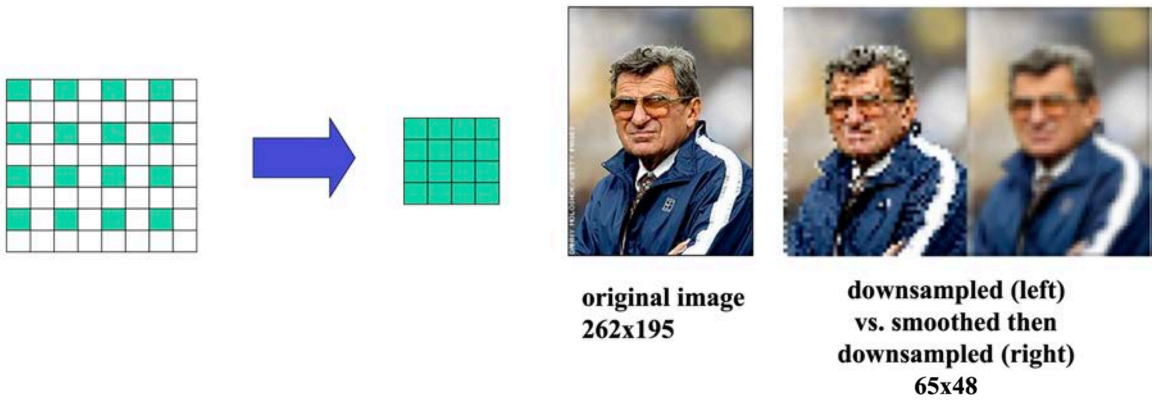


Figure 2.2: The general idea of image down-sampling. Downsampling over a previously blurred(smoothed) image to avoid aliasing.

## Image Subtraction and Difference of Gaussians (DoG)

The reason for image subtraction is that Laplacian of Gaussian causes intensive computation. The difference-of-Gaussian function provides a close approximation to the scale-normalized Laplacian of Gaussian, $\sigma^2\nabla^2G$, as studied by Lindeberg (1994). The derivative calculation is thus simplified to subtraction of adjacent images.

When we subtract one image from another, the result is a new image representing the differences between the two source images. For the same pixel on the corresponding location of source images, the result is 0, which is represented by a pure black pixel on the new image. If the corresponding pixels are different, the result is a value that represent how much they differ. We get some pixels closer to white as a result of greater difference and pure white for binary images, as shown in Figure 2.3. [10]

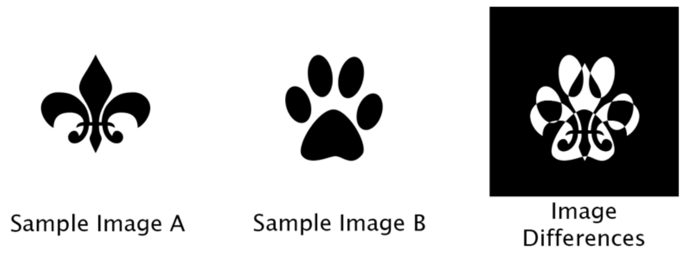


Figure 2.3: An intuitive illustration for image subtraction. Black for same pixels and white for difference.

Having this way of image subtraction, the scale difference due to different sigma value can be found by applying this method on a pair of images that only differs in sigma value. After the subtraction, we should have the features which are visible in the image with lower sigma value but not visible with the higher sigma. In other words, these features are the details that only visible at a given scale. Subtracting each pair of consecutive scales (blurred images) inside each octave, another set of images namely the Difference of Gaussians (DoGs) are thus generated.

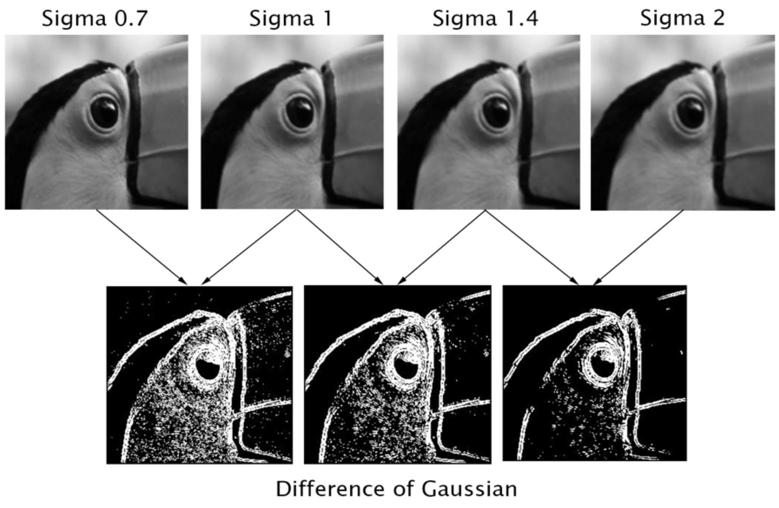


Figure 2.4: The concept of finding features at a given scale by subtracting images with different sigma.

However, why do we "coincidentally" get exact the outline sketch? The magic here is that the Gaussian blur "smooths" out the outline of the images. Areas that have very little change in contrast appear similarly, despite one image having a stronger blur than the other. Imaging we have a picture for a piece of white paper on a desk, it looks almost the same for the entire white part when we change the sigma value. But the edge of this paper changes relatively a lot when it becomes more blurred. Since there is little change in the area of the paper, the result of the subtraction is closer to zero, which turns to be black. In high contrast areas, we can surely get some white lines, which indicates the greater value in result of this subtraction, corresponding to the edge of the paper because of the high contrast on the edges. The greater the contrast, the more resilient the area is to a Gaussian at a lower sigma, and the more details to lose as increasing the blur effect.

According to Lowe's research [7], the stability of keypoint detection increases with a higher sigma. But using a larger sigma reduces the efficiency, and they chose sigma equals 1.6 as a trade-off.

**From DoG to Feature**

We have a stack of images with Gaussian blurs at different scales, and these images are sorted based on the scale value, i.e. the sigma value. The highest scale matches the smallest image having regard to pixel resolution. As a result, it would be less time-consuming to do coarse-to-fine searches, i.e. to start from the image with smaller pixel resolution.

Illustrated in Figure 2.5, we compare each pixel against the eight neighbors in the same scale and nine neighbors in the next and previous scales. If our pixel is still the local extrema when compared to all of its closest neighbors across three scales, then we have an x and y position for this feature along with its scale value. [10]

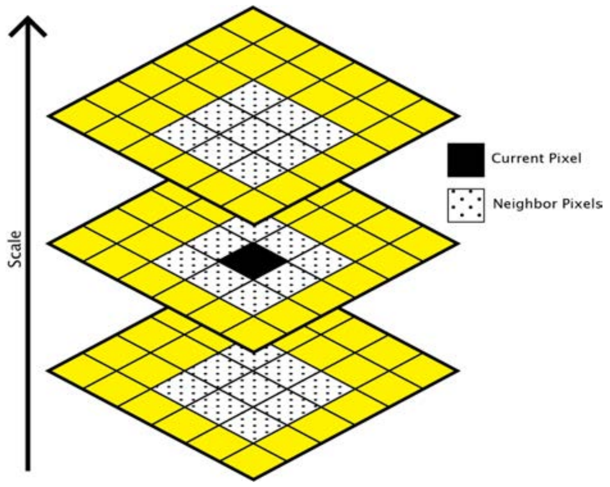


Figure 2.5: The current pixel and its adjacent scales after Differences of Gaussian.

However, not all of these extrema are the so-called features. Some of them, which do not tend to be robust, have to be weeded out determined by certain threshold. Although edge features seem to be great as we want to detect some slender body, e.g. cranes and scaffolding, this can be harmful due to the lack of stability. Small amounts of noise would have an influence on these edge features, because they sometimes do not have an accurately determined location. After low-contrast and edge features are filtered, the result till now should be a robust list of features which include scale.

### 2.1.2 Feature Description

#### Keypoint Orientations

Having the relatively stable features filtered by thresholding, the next step is to assign an orientation, which provides rotation invariance, to each point. Due to the different scale, at which these features are captured, each feature seems to have different range of the "orientation collection region", if we measure it by the content of the image in this region. Because the gradient is calculated pixel-wise, this collection region around the keypoint is bigger (measured in percentage) for bigger scale.

The results of gradient direction and magnitude are filled into a histogram as shown in Figure 2.6 [11], where the abscissa axis represents the degree in regard to each point with even spacing. 360 degrees are divided equally into 36 sectors, each of which takes the degree value as orientation section and shows up in the histogram with its magnitude. From the histogram, it is then easy to find out the peak at the corresponding bin. And any peaks above 80% of the highest peak are converted into a new keypoint, having the same location and scale as the original.

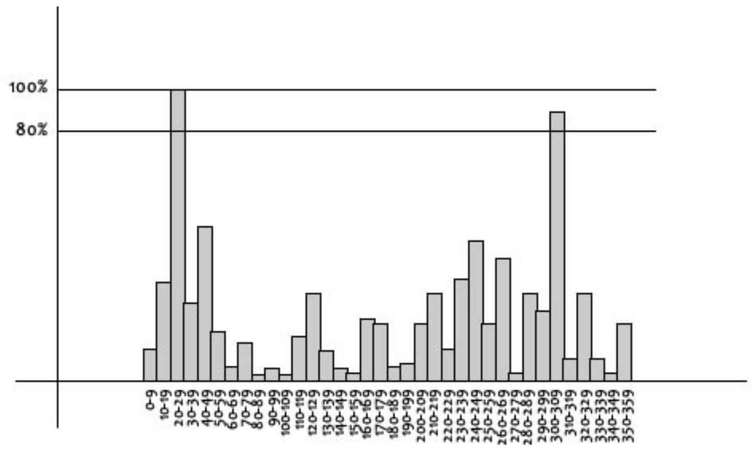


Figure 2.6: the histogram representing orientations and corresponding magnitudes.

In this way, the most prominent orientation(s) in the surrounding region is figured out and assigned to the keypoint to guarantee the rotation invariance.

**Keypoint Descriptor**

The keypoint descriptors are generated based on location, scale and orientation to each keypoint and thus are highly distinctive. In this way, the descriptor can be considered as a unique Fingerprint for each Keypoint. Each single feature uses this fingerprint to match with the feature in another image which is stored in a large database of features.

Colmap uses 4x4 windows around the keypoint, each of which is considered as a circle divided into 8 equal parts. This generates a 4x4x8, namely 128-dimensional vector. In fact, a keypoint does not lie exactly on a certain pixel, instead, it lies "in between" pixels.

### 2.1.3 DSP-SIFT

Domain-size-pooled SIFT, or DSP-SIFT [12] is a relatively simple modification of local image descriptors, obtained by pooling gradient orientations across different scales, in addition to spatial locations. In the experiments by J. Dong et al., DSP-SIFT outperforms the best CNN which has a much larger dimension. The new terminology — pooling, is commonly understood as the combination of responses of feature detectors/descriptors at nearby locations, aimed at transforming the joint feature representation into a more usable one that preserves important information (intrinsic variability) while discarding irrelevant detail (nuisance variability). [13, 14]

In the following Figure 2.7, the difference between SIFT and DSP-SIFT is shown. In SIFT (top, recreated according to [7]) isolated scales are selected (a) and the descriptor constructed from the image at the selected scale (b) by computing gradient orientations (c) and pooling them in spatial neighborhoods (d) yielding histograms that are concatenated and normalized to form the descriptor(e). While in DSP-SIFT (bottom), pooling occurs across different domain sizes (a): Patches of different sizes are re-scaled(b), gradient orientation computed (c) and pooled across locations and scales (d), and concatenated yielding a descriptor (e) of the same dimension of ordinary SIFT.

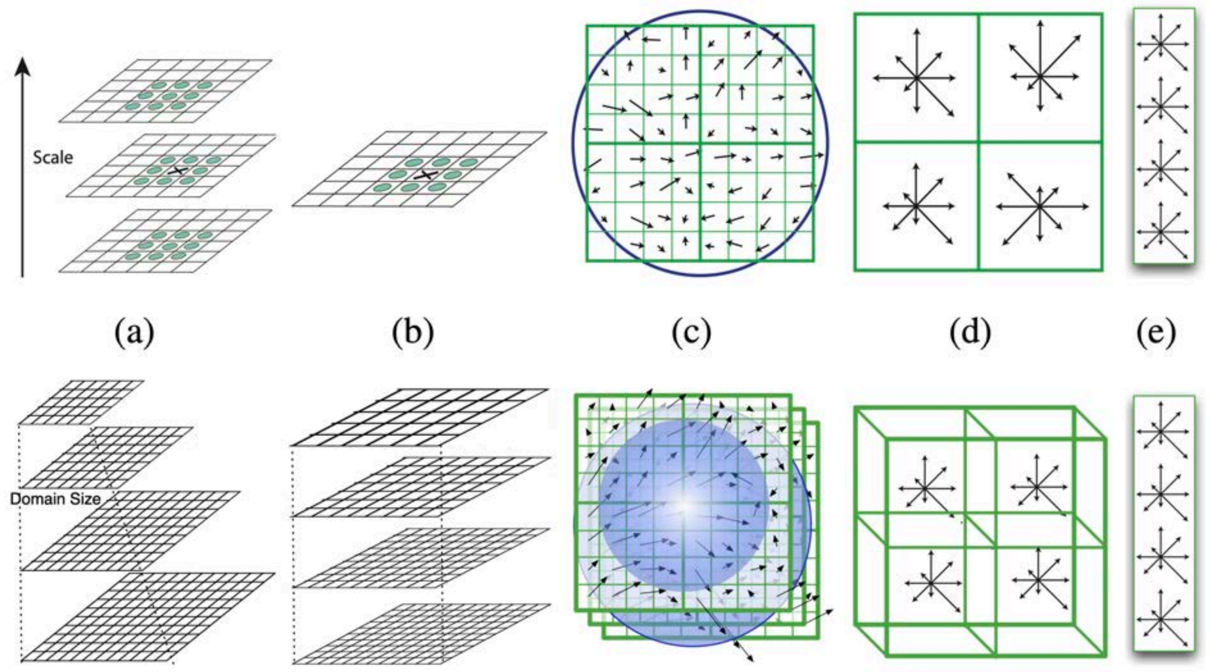


Figure 2.7: DSP (bottom) in comparison with normal SIFT (up).

There are more parameters introduced by DSP-SIFT and some are later adopted in Colmap. For a detected scale, DSP-SIFT samples scales within a neighborhood with coefficients for this scale. And the lower- and upper-bound are called *SiftExtraction.dsp_min_scale* and *SiftExtraction.dsp_max_scale* as optional flags in *feature_extractor* of Colmap. As their experiments [12] show, the number of size samples to construct DSP-SIFT provides worthy improvement until 10. And that corresponds with *SiftExtraction.dsp_num_scales* in Colmap. A full configuration over DSP is included in Appendix A.

## 2.2 Incremental Structure-from-Motion

Incremental Structure-from-Motion is a prevalent strategy for 3D reconstruction from unordered image collections. Colmap adopts a new SfM technique [5] to make a further step towards the goal of truly general-purpose SfM system for better robustness, accuracy, completeness and scalability.

SfM is the process of reconstructing 3D structure from its projections into a series of images taken from different viewpoints. Incremental SfM is a sequential processing pipeline with an iterative reconstruction component, as shown in Figure 2.8. This whole process sets up incrementally a reliable scene graph from unordered image collection, serves as the foundation for the reconstruction, and is followed by refinements as triangulating scene points, filtering outliers and bundle adjustment (BA). [5]

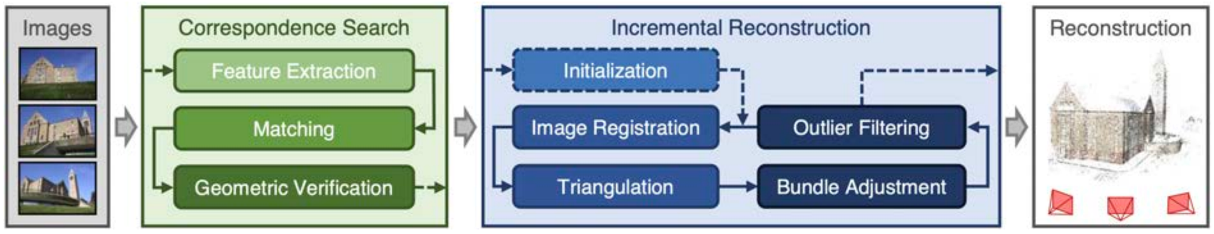


Figure 2.8: Incremental Structure-from-Motion pipeline.

The process of SfM can be divided into two main parts, corresponding to the Figure 2.8. Firstly, invariant features are to find and SfM recognizes them in multiple images which are then referred to overlapping images. Here, SIFT plays one of the most important roles in the standard of extraction. The features, as appearance descriptions of images, are then matched up by searching for similarity of appearance. The feature correspondences are associated to the set of potentially overlapping image pairs. At last, the geometric verification is executed by trying to estimate a transformation that maps feature points between images using projective geometry. If a valid transformation maps a sufficient number of features between the images, they are considered geometrically verified. The output of correspondence search is a scene graph with images as nodes and verified pairs of images as edges.

The second part, incremental reconstruction starts with a carefully selected two-view reconstruction [15] for initialization. Choosing a suitable initial pair where many overlapping cameras shoot towards results in a more robust and accurate reconstruction due to increased redundancy. More images are then added to this growing model by solving the Perspective-n-Point (PnP) problem using feature correspondences to triangulated points in already registered images. This PnP problem is to estimate the pose of a calibrated camera, with a given set of 3D points in the world and their corresponding 2D projections in the image. For uncalibrated cameras, the intrinsic parameters can be used. By Colmap a novel method called Next Best View Selection for pose estimation was proposed. After image registration comes the triangulation, adding new scene points, for which the scene part is also covered from a different viewpoint by registered images. Triangulation increases the stability of the existing model through redundancy [16] and enables registration of new images by providing additional 2D-3D correspondences. Finally, the iterative bundle adjustment provides further refinement, minimize the reprojection error and use a loss function to potentially down-weight outliers.

This incremental SfM generates and augments the scene graph with improving robustness and accuracy, establishes a complete and precise model in its coordinate system as sparse model of Colmap, based on which the dense model and point-cloud are created.

## 2.3 Multi-View Stereo

Multi-View Stereo system provides a robust, accurate and efficient dense modeling from unstructured image collections. MVS leverages multiple views to overcome the inherent occlusion problems of two-view approaches. [17] The method adopted in Colmap simultaneously considers a variety of photometric and geometric priors improving upon the robustness and accuracy of the depth estimation framework by Zheng et al. [19].

Different applications may use different implementations, but the overall approach is similar: Collect images, Compute camera parameters for each image, reconstruct the 3D geometry of the scene from the set of images and corresponding camera parameters, optionally reconstruct the materials of the scene. [27] Benefit from the success of SfM, the workload in MVS part is significantly reduced and the accuracy is already calibrated.

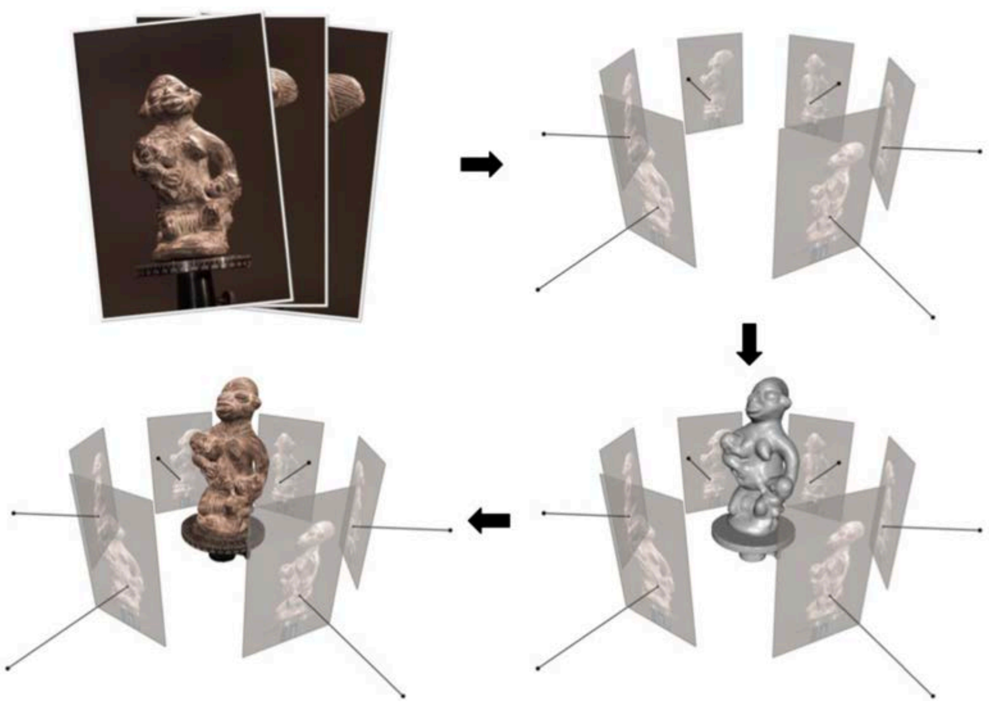


Figure 2.9: Example of a multi-view stereo pipeline. Clockwise: input imagery, posed imagery, reconstructed 3D geometry, textured 3D geometry.

In application Colmap [18], Multi-View Stereo (MVS) takes the output of SfM to compute depth and/or normal information for every pixel in an image. Fusion of the depth and normal maps of multiple images in 3D then produces a dense point cloud of the scene. The method performs MVS with pixelwise view selection for depth/normal estimation and fusion.

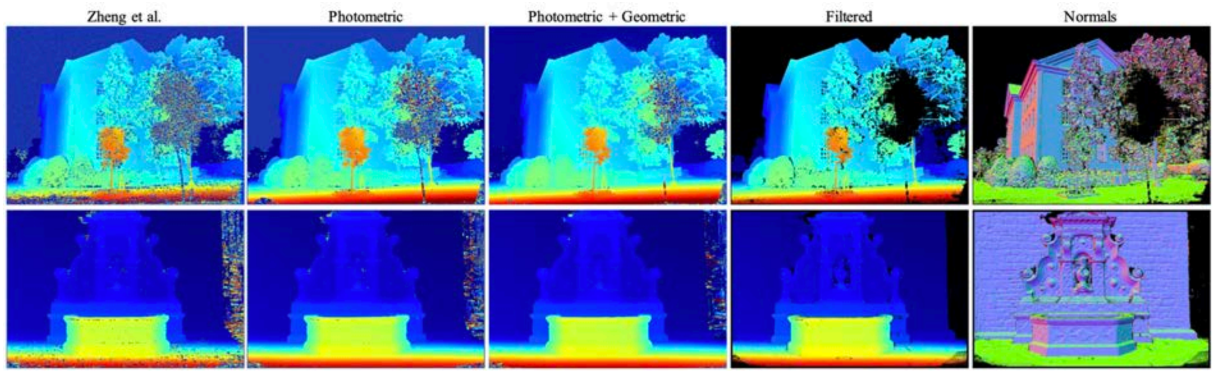


Figure 2.10: Reconstruction results shown in [17].

As Colmap infers the best depth and normal based on both photometric and geometric consistency in multiple views, it generates `image_name.JPG.photometric.bin` and `image_name.JPG.geometric.bin` under `stereo/depth_maps` corresponding to each image by default. To figure out why there are two kinds of depth maps, here are some supplementary information about photometric and geometric consistency.

A product of the illumination flux reaching the surface from the light source and albedo is the observed intensity. As written in Wikipedia [20], the apparent brightness of a Lambertian surface to an observer is the same regardless of the observer's angle of view. [21] That means the intensity is independent from the viewing direction.

An image point in view one that corresponds to an image point in view two are photometrically consistent, if their intensity difference corresponds to image noise, motivating measuring sum of squared differences (SSD) or normalized cross correlation (NCC) between fixed windows around candidate corresponding points. [22]

The idea behind the geometric consistency check is to scrutinize the images if the matches are compatible with the expected geometric transformation. If a significant fraction of matches is known to be incorrect, the reliability can be greatly improved by enforcing geometric consistency constraints. [23] There are two general strategies: (1) estimate the geometric transformation by applying an (usually robust) estimator, and eliminating the matches incompatible with that transformation; (2) computing a statistical distribution of some geometrical transformation parameter (rotation, scale) of each match, and eliminating the matches which deviate too much from the mode of that distribution.

As Schönberger et al. [17] pointed out, a popular approach to filter outliers due to noise, ambiguities, occlusions, etc. is to enforce multi-view depth coherence through left-right consistency checks as a post-processing step [24, 25]. They integrate multi-view geometric consistency constraints into the inference to increase both the completeness and the accuracy. More specific, it's to compute the geometric consistency between two views as the forward-backward reprojection error $\psi_l^m$. And the estimated depths are consistent if the reprojection error $\psi_l^m$ is small. The value of this reprojection error is adjustable in Colmap with flag *--PatchMatchStereo.filter_geom_consistency_max-_cost* and *--StereoFusion.max_reproj_error*. According to [17], their experiments demonstrate improvements on both completeness and accuracy.

However, according the experiments from [26], geometric consistency alone may be unable to guarantee high-quality results in databases that contain too many non-discriminating descriptors. In their second study, a large number of non-discriminating descriptors are spawned by problematic image features like tree branches and complex shadows, depicted in Figure 2.11. Their voting algorithm with the RANSAC doesn't lead to bad results, but the impact from geometric consistency is much subdued.

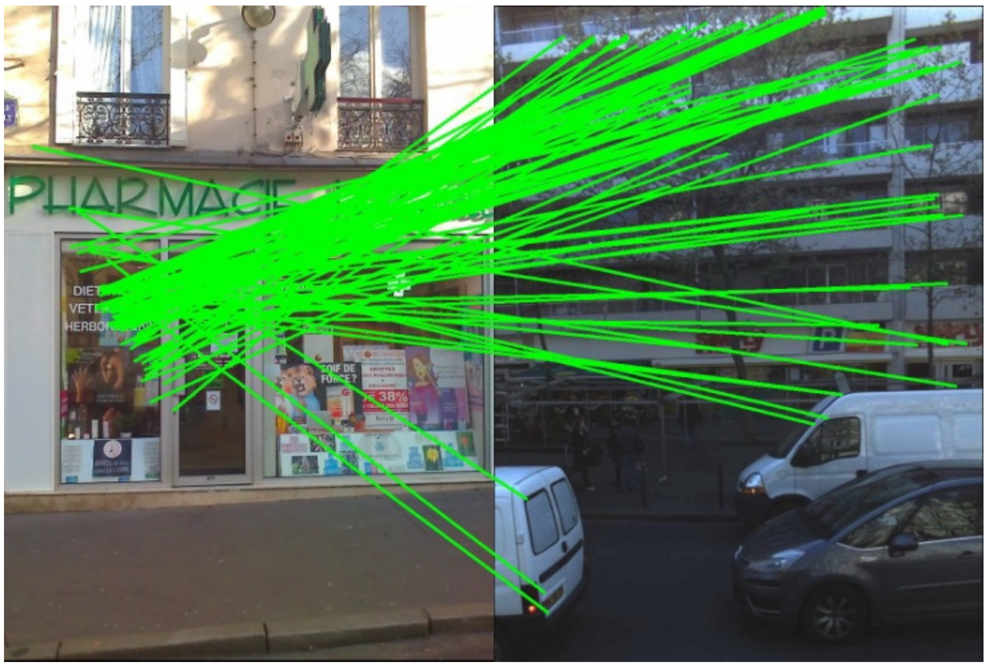


Figure 2.11: Non-discriminating features like from tree branches disturb the matching.

In our study, there are some similar situations like the scaffoldings and tree branches, also the tower crane has a very thin structure with its complex shadows. That's a reason why geometric consistency maybe not deliver an ideal effect as expect. Besides, if the reconstruction doesn't benefit from this, it would cause unnecessary waste of time.

# 3 Methodology

This chapter first introduces the involved software and some commonly used commands. Then the interactive methods are set forth, including how to personalize Colmap and how to process its critical output — depth maps. Meanwhile, we can gain some necessary knowledge over image process and OpenCV Library. Afterwards, how to evaluate depth map based on the benchmark of hand-labelled image is explained. At last, the concept of optimization is proposed. With the following tools, as well as additional Python scripts, the whole study method can be strung together.

## 3.1 Toolkit

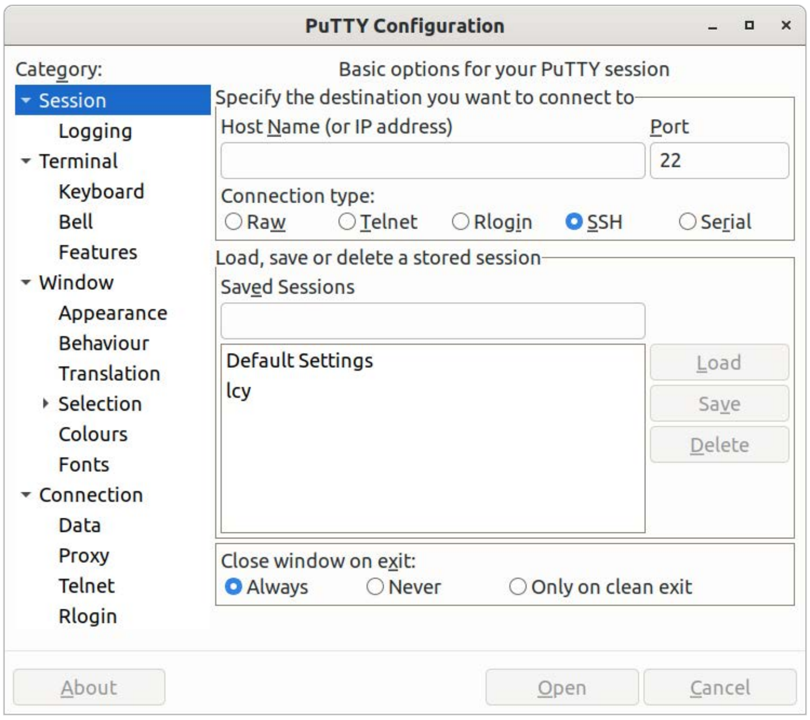
### 3.1.1 Connection with Server

The computation for over hundred full resolution images requires high performance on both CPU and GPU, it also generates many big chunks of data. Fortunately, there is a high-performance server provided by Chair of Computational Modeling and Simulation (CMS), which provides full support for CUDA and high-speed computation. We can thus stably process full resolution images and keep more data until optimization finishes.

Lack of display on the server, data such like images and point-cloud cannot be directly viewed. Thus, PuTTY Client and SFTP in terminal are used to interact with server. Providing public SSH key to the server and keeping private SSH key on local computer, a remote control is established.

PuTTY SSH Client is a TTY used to access the data on the server and carry out instructions. It can save the custom settings for IP address and font, then load from previous. After setting up, it works almost the same as a normal terminal.

SSH File Transfer Protocol (SFTP) is a network protocol, which is used to transfer data between server and local computer in this study. Integrated in command line, it's very useful to use *put* to upload a configuration file and download the results to local using *get*, and *mkdir* can also be used to create a new directory. A difference to notice, this tool has very limited functionality compared with using PuTTY. For example, *get directory_from_server -r* won't work, because the flag *-r* has to be directly after *get*.



Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address): 
Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

Default Settings
lcy

Close window on exit: [✓ Always] Never Only on clean exit

Figure 3.1: The interface of PuTTY before opening.

There are some other small tools such as htop and nvidia-smi. By typing *htop* in a command line we can check the process status of the server, because the server is shared by multi-user. And *nvidia-smi* is for checking GPU status. In case the server being overloaded, the calculation time would be prolonged at times.

## 3.1.2 Docker

Docker is the platform we use to build and run applications within containers. Fundamentally, a container is nothing but a running process, with some added encapsulation features applied to it in order to keep it isolated from the host and from other containers. One of the most important aspects of container isolation is that each container interacts with its own private filesystem; this filesystem is provided by a Docker image. An image includes everything needed to run an application - the code or binary, runtimes, dependencies, and any other filesystem objects required. [28]

### Dockerfile

Although the server doesn't have Colmap installed, we may execute Colmap inside a container. In this way, it's also able to execute several reconstructions by running Colmap in different containers. And their data are isolated in each own storage. This feature is important for optimization, as we usually don't want to overwrite the one reconstruction after altering some parameters. Instead, we need to keep each reconstruction model separated in order to have their results restored to compare.

Hence, a personalized Docker image is required. Like many other Docker images, our Docker image for Colmap is also based on a parent image. Due to the need of CUDA, *nvidia/cuda:10.2-cudnn7-devel* is chosen. This parent image is optimal for its stable CUDA 10.2 and also due to its *gcc* version of 7.4, which is lower than 8 and thus avoid incompatible compilation of Colmap. Although an executable Colmap can be installed like running *sudo apt install colmap* in a terminal, a complete version of the newest Colmap has to be installed by following exactly the same steps as shown in its installation documentation. The price of building from source code is the long compiling time. An idea to prevent building from source code is to set this personalized image as another parent image, and any new functionality for a Docker image can be easily added based on this new parent image.

The Dockerfile, which is used to build the image for this thesis, is provided in Appendix B.1 and also available in DockerHub:

*https://hub.docker.com/repository/docker/ripfreeworld/colmap_cuda10.2*

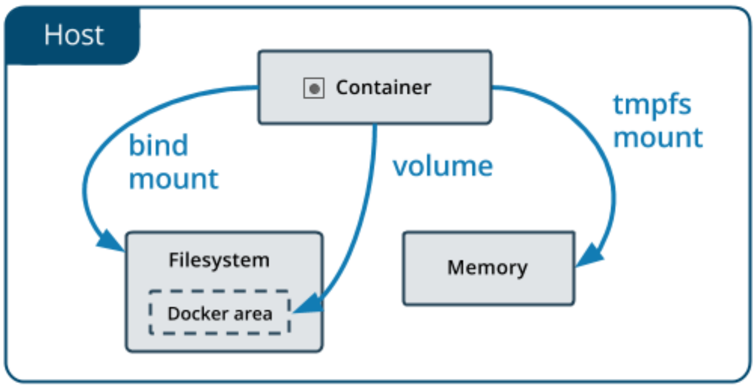


Figure 3.2: Volumes and bind mount.

## Storage

By default, all files created inside a container are stored on a writable container layer, where the data doesn't persist when this container no longer exists, and it can be difficult to get the data out of the container if another process needs it, or we want to view some partial results. As solution, Docker provides two main options for containers to store files in the host machine, so that the files are persisted even after the container stops: volumes and bind mounts, as shown in Figure 3.2. [29]

Although volumes have several advantages over bind mounts, we neither often share the volume among multiple containers nor encrypt the contents of volumes, it's rather intuitive to directly bind mount the folder into the server so that the images or the depth maps can be accessed and viewed easily on the local computer, due to the lack of monitor for the server. And the file and directory structure of the Docker host is guaranteed to be consistent with the bind mounts the container requires, in this case the bind mounts is appropriate.

Bind mount can be used by adding a flag *-v*, e.g. *-v /path/in/system/:/path/in/container.*

**Some Other Important Flags**

In addition to *-v* for bind mount, here are some necessary flags to append while running a Docker container in this study.

*--gpus all* is appended to explicitly ask for GPU enabling. More delicate usage instead of *all* can be found in NVIDIA/nvidia-docker [30].

*-d* stands for detach, i.e. daemon mode. That means the container would initialize and keep working in backstage. The container can be accessed by *docker attach container_name*. It doesn't matter to close the current window, even when some instructions are running. This daemon mode is very handy for time-consuming cases, e.g. *patch_match*.

*--name* for an explicit name rather than a random name to let others on the server know from whom and about what this container is.

*-it* is used for interactive TTY. It's a pseudo terminal, without which we cannot send inputs to the container interactively.

**Different Privilege for Files generated from Docker**

Additionally, the data generated by the Docker via bind mounts cannot be deleted directly in command line. Docker is started with root privilege, while using the command line via PuTTY has a normal user privilege. The data are strongly connected by colon characters ( : ) as key value pair, e.g. *-v /my/local/folder:/folder/in/docker*. When something is generated inside the Docker, they are synchronized on the local implicitly by root. So, we are not able to touch this data anymore because of a lower authority.

**Docker-Compose**

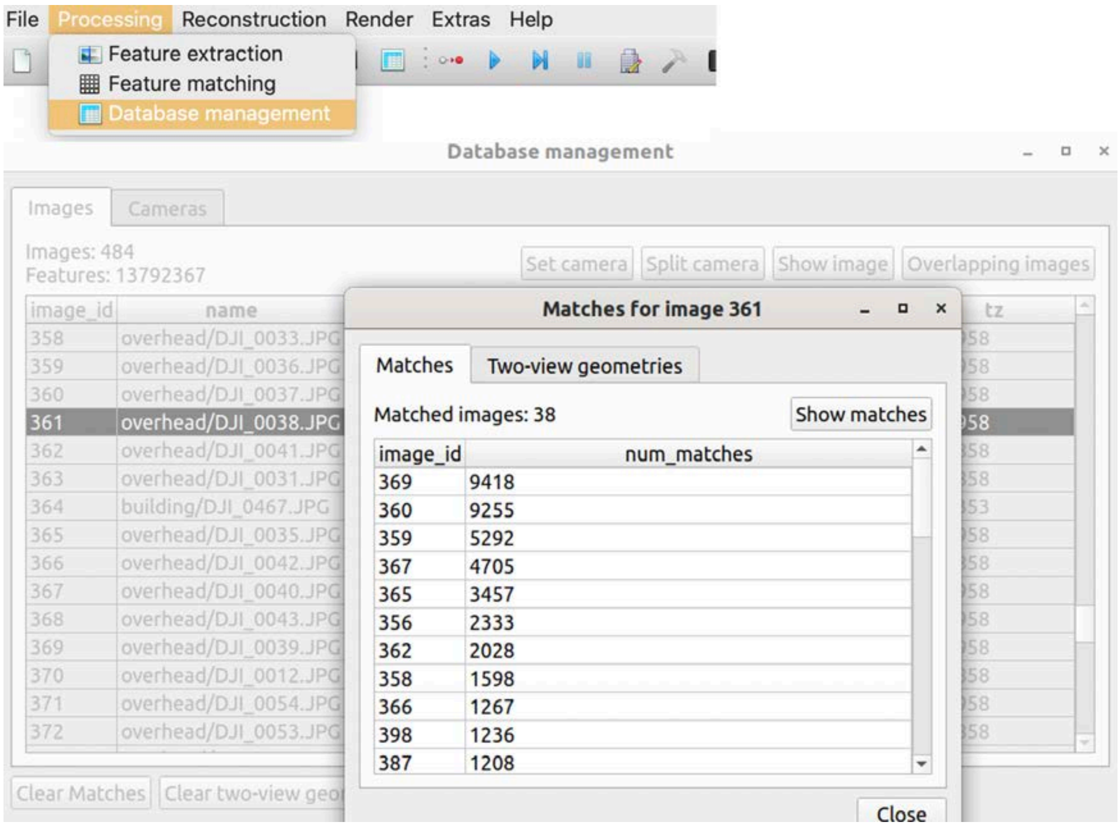
There is an integrated version for multiple containers executing on a same dataset — docker-compose, which is built up by a 'yaml' file to configure services. The `Dockerfile` and the flags above can all be included in a `docker-compose.yml` file. Although most of its usage is related to network, we can benefit from docker-compose by attaching a high-quality optimization algorithm to the reconstruction.

An iterative optimization is based on the same model from reconstruction. If the optimization is "intelligent" enough which takes a long time to evaluate and analyze, it means the reconstruction part has to idle until evaluation finishes. For instance, an evaluation process needs to read many depth maps and detect the object in order to compare with existing data, this could take a long time. By using docker-compose, multiple containers can work simultaneously. That means while evaluating one result, reconstruction can go for a next parameter combination. And there could be another container responsible for storing data and logs into a database, as well as preventing possible file locking by updating status. Separating the containers for different functionality makes the program “drier”. And data can be shared inside this docker-compose via *volumes* by an internal network-bridge.

### 3.1.3 Colmap

COLMAP is a general-purpose Structure-from-Motion (SfM) and Multi-View Stereo (MVS) pipeline with a graphical and command-line interface.[31] It's the Software we choose to generate 3D model from images. Related to the theories, Colmap detects keypoints in each image whose appearance is described by numerical descriptors. Pure appearance-based correspondences between keypoints/descriptors are defined by matches, while inlier matches are geometrically verified and used for the reconstruction procedure. [32]

The GUI version of Colmap is well packaged, providing most of the available functionality and visualizes the reconstruction process in real-time. However, the GUI application requires an attached display, which is not available on the server. While the GUI is more interactive, CLI provides a more straight-forward and efficient execution for full functionality. In this thesis, we mainly focus on the command line version. Each work stage is subdivided more separately and Colmap is thus able to connect with the parameter parser explained in the next chapter.



Images: 484
Features: 13792367

| image_id | name |
|---|---|
| 358 | overhead/DJI_0033.JPG |
| 359 | overhead/DJI_0036.JPG |
| 360 | overhead/DJI_0037.JPG |
| 361 | overhead/DJI_0038.JPG |
| 362 | overhead/DJI_0041.JPG |
| 363 | overhead/DJI_0031.JPG |
| 364 | building/DJI_0467.JPG |
| 365 | overhead/DJI_0035.JPG |
| 366 | overhead/DJI_0042.JPG |
| 367 | overhead/DJI_0040.JPG |
| 368 | overhead/DJI_0043.JPG |
| 369 | overhead/DJI_0039.JPG |
| 370 | overhead/DJI_0012.JPG |
| 371 | overhead/DJI_0054.JPG |
| 372 | overhead/DJI_0053.JPG |

Matched images: 38

| image_id | num_matches |
|---|---|
| 369 | 9418 |
| 360 | 9255 |
| 359 | 5292 |
| 367 | 4705 |
| 365 | 3457 |
| 356 | 2333 |
| 362 | 2028 |
| 358 | 1598 |
| 366 | 1267 |
| 398 | 1236 |
| 387 | 1208 |

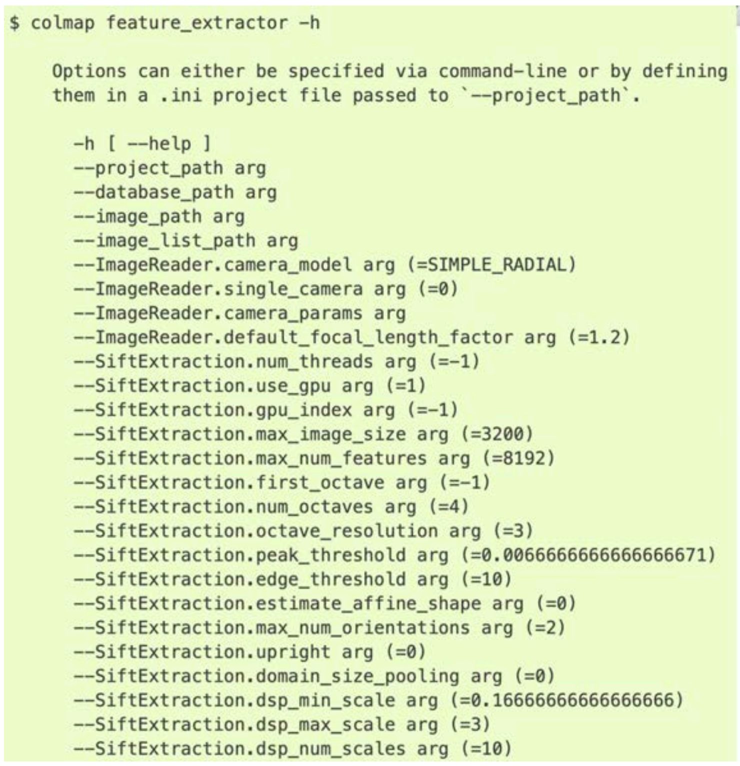
Figure 3.3: Query matched overlapping images for a certain image via the GUI.

Additionally, we use the GUI version of Colmap to easily query overlapping images for a certain image in database. A mapping from `image_id` to image name can be acquired from its database management as shown in Figure 3.3. The matched images are sorted by the number of matched features in descending order.

### 3.1.4 Parameter Parser for Colmap

Colmap provides Command-line Interface, where more available options can be given by delivering diverse flags. However, in our case, more customized parameter assignments are needed than usual situation. When only few parameters to be changed to have a new reconstruction, it would be very verbose to assign again each customized but not-to-be-changed parameter.

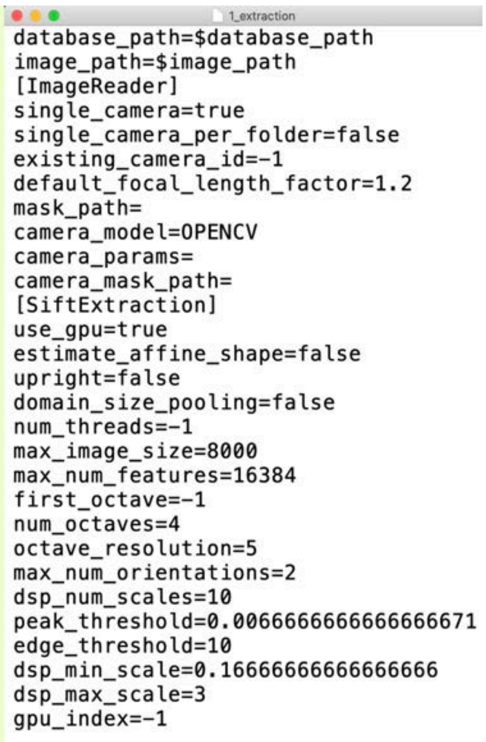
Fortunately, Felix Eickeler, from the chair of CMS, provides a handy application for reading and parsing the parameters from text files. Instead of typing instructions in CLI, the wanted job can by assigned and executed by running this Python file. With this parser application, it's possible to alter a parameter by passing the value inside the Python script instead of appending flags in command-line.



```
$ colmap feature_extractor -h

    Options can either be specified via command-line or by defining
    them in a .ini project file passed to `--project_path`.

      -h [ --help ]
      --project_path arg
      --database_path arg
      --image_path arg
      --image_list_path arg
      --ImageReader.camera_model arg (=SIMPLE_RADIAL)
      --ImageReader.single_camera arg (=0)
      --ImageReader.camera_params arg
      --ImageReader.default_focal_length_factor arg (=1.2)
      --SiftExtraction.num_threads arg (=-1)
      --SiftExtraction.use_gpu arg (=1)
      --SiftExtraction.gpu_index arg (=-1)
      --SiftExtraction.max_image_size arg (=3200)
      --SiftExtraction.max_num_features arg (=8192)
      --SiftExtraction.first_octave arg (=-1)
      --SiftExtraction.num_octaves arg (=4)
      --SiftExtraction.octave_resolution arg (=3)
      --SiftExtraction.peak_threshold arg (=0.0066666666666666671)
      --SiftExtraction.edge_threshold arg (=10)
      --SiftExtraction.estimate_affine_shape arg (=0)
      --SiftExtraction.max_num_orientations arg (=2)
      --SiftExtraction.upright arg (=0)
      --SiftExtraction.domain_size_pooling arg (=0)
      --SiftExtraction.dsp_min_scale arg (=0.16666666666666666)
      --SiftExtraction.dsp_max_scale arg (=3)
      --SiftExtraction.dsp_num_scales arg (=10)
```



```
database_path=$database_path
image_path=$image_path
[ImageReader]
single_camera=true
single_camera_per_folder=false
existing_camera_id=-1
default_focal_length_factor=1.2
mask_path=
camera_model=OPENCV
camera_params=
camera_mask_path=
[SiftExtraction]
use_gpu=true
estimate_affine_shape=false
upright=false
domain_size_pooling=false
num_threads=-1
max_image_size=8000
max_num_features=16384
first_octave=-1
num_octaves=4
octave_resolution=5
max_num_orientations=2
dsp_num_scales=10
peak_threshold=0.0066666666666666671
edge_threshold=10
dsp_min_scale=0.16666666666666666
dsp_max_scale=3
gpu_index=-1
```

Figure 3.4: Left is the Colmap command line interface, right is the parameter list for parser app.

In *feature_extractor* stage for example, Colmap provides many parameters, most of which are in 2-part hierarchic series. It's troublesome and error-prone to type these long flags in command line. With the parser program, text file breaks the hierarchy and classify these parameters under *[ImageReader]* and *[SiftExtraction].* Each variable name on the right-hand side after a dollar sign '$' is to substitute from *class ReconstructionConfig* by its constructor inside the Python script, so that the value can be altered in run-time.

The full source code of this parameter parser application doesn't appear in the Appendix, since it's not the original author.

### 3.1.5 CloudCompare

CloudCompare is an open source 3D point cloud processing software, which can display large dense point-cloud smoothly and even perform comparison between two point-clouds. In this study, we mainly use this software to check the fused output from Colmap. Compared with another software MeshLab is CloudCompare in this study apparently faster. And its optional rotation center is very handy since the main content of point cloud locates usually not in the center of the coordination due to some big outliers.

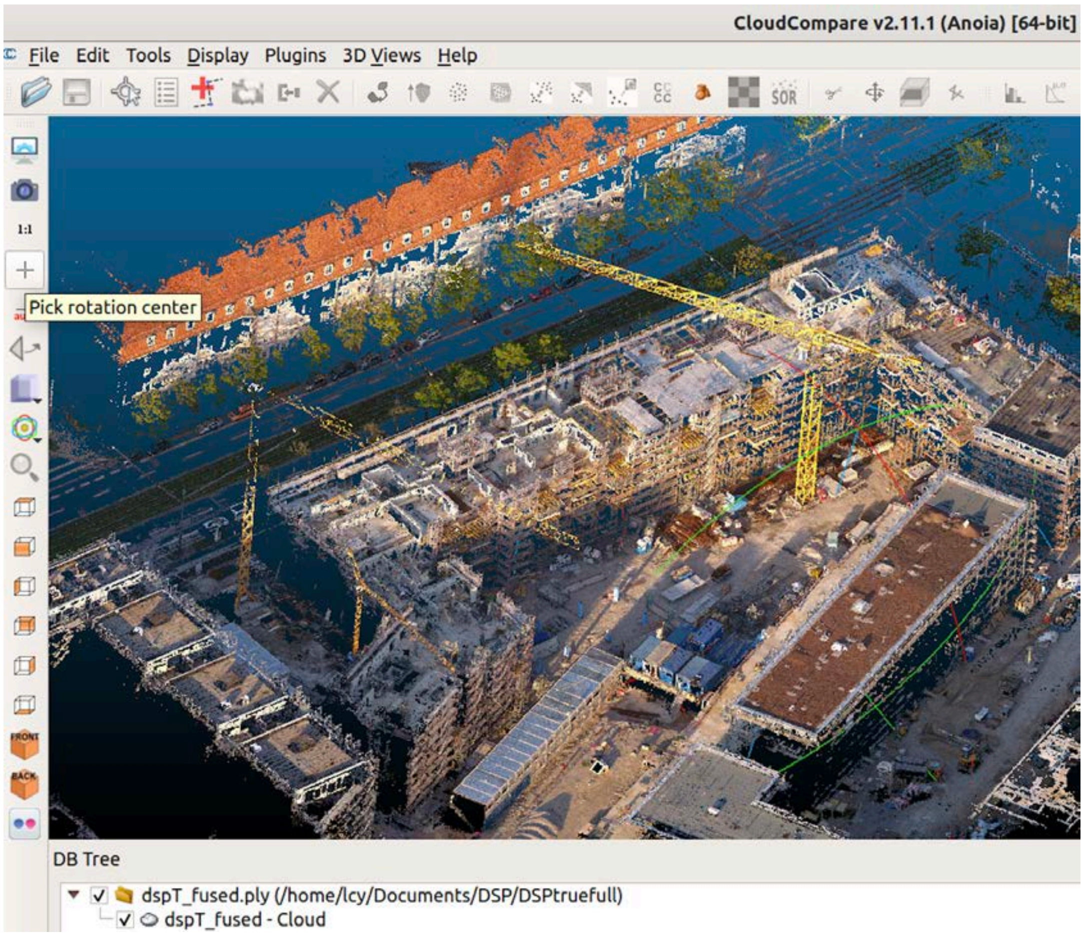

CloudCompare v2.11.1 (Anoia) [64-bit]
File Edit Tools Display Plugins 3D Views Help
Pick rotation center
DB Tree
dspT_fused.ply (/home/lcy/Documents/DSP/DSPtruefull)
dspT_fused - Cloud

Figure 3.5: The interface of CloudCompare and its functionality "pick rotation center".

## 3.2 Configuration and Customized Reconstruction

The dataset for experiments consists of two sets of images obtained by handheld shooting on the ground and by a drone respectively. They can be fused into one continuous coordinate system, while a smaller subset for experiment, mentioned later in Chapter 4.3, is selected purely out of the overhead images. The evaluation and optimization based on the quality of tower crane focus mainly on the smaller dataset from overhead.

According to Colmap and the parser application, the work process is divided into 8 stages: feature extraction, exhaustive matching, mapper, bundle adjustment, model alignment, image undistortion, patch match and stereo fusion. For this dataset there is no need to geo-register model to coordinate system, the *model_aligner* is thus unused. The first four stages are for sparse model and the rest for dense model. *image_undistorter* provides undistorted images with resolution slightly different from the original source images. The most evaluation and optimization in chapters afterwards uses the undistorted images for labelling and comparison.

We use the automatic reconstruction of Colmap using its defaults as a fully non-optimized version. As for optimization for the complete dataset, we first adjust the parameters relating to hardware information like *num_threads* and *cache_size* to a moderate value. And resolution of images in feature extraction, image undistortion and patch match stages is set to max size or a relatively high value to cover full-size images. *-1* indicates the maximal available value in configuration.

The optimization begins with SfM, i.e. sparse model. By turning on the DSP and increasing search range for more features like raising *max_image_features* and *num_octaves*, the algorithm using great computing power from the server to find as many features as possible in full resolution images. A sparse model with high precision and more detail features provides a robust fundament in camera position and feature points, which act as input for the following dense model. We use the pair of DSP parameters mainly as advised by J. Dong et al [12].

In our case, the number of images in the dataset is relatively low (up to several hundreds), this matching mode should be fast enough and leads to the best reconstruction results. [33] In exhaustive matching, every image is matched against every other image, while the block size determines how many images are loaded from disk into memory at the same time. With a big RAM of the server, we set the *block_size* to *120*.

Instead of listing every stage, we only focus on the patch match which is later iteratively computed for evaluation and optimization. Patch match stage plays the important role in dense points determination and decide whether the point-cloud is more detailed in fine structures or smoother and more continuous. In this stage, we can ask for a filtered version by setting *filter=true* and decide the criteria such as *filter_min_ncc* for minimum NCC coefficient. And we can also turn off the geometric consistency check by setting *geom_consistency=false*. *window_radius* is a critical parameter to balance efficiency and quality; we will check if it's worth the potential improvement in quality for our case over construction site. In following chapters, we analyze the difference of depth-maps generated by photometric and geometric consistency check and test the depth-maps under different size of *window_radius* by using the proposed evaluation method.

## 3.3 Process on Depth-Map

Point-cloud, as the final output, contains the whole information of the reconstruction. The purpose of the optimization of reconstruction is to get a more precise model, namely a precise spatial relationship of the significant points. Although we can tell some of the difference between two point-clouds through comparing in CloudCompare, usually it is hard to judge which one is better, especially when only few of the parameters in reconstruction phase are modified with only small step size.

After Patch Match stage, Colmap provides depth-maps under `/dense/stereo/-depth_maps` folder, which are very useful for reflecting a precise spatial relationship in the point cloud. Comparing two images becomes easier, and comparing two binary black-white (no gray) images is the easiest. The mask for tower crane generated from depth-map and the mask for ground-truth are to compare. Minimizing the numerical error of the comparison serves as the objective. Hence, this chapter provides a straight-forward evaluation method on the quality of depth-map, by calculating the precision and recall by bitwise comparing the masks of depth-maps with the corresponding ground truth generated based on the representative undistorted image DJI_0038.JPG.

### 3.3.1 Information behind Image

First of all, understanding the information behind image format is necessary to deal with images. In fact, all images consist of numbers in form of multi-dimensional matrix. And the resolution is the size of this matrix. Colorful images typically have RGB color model representing red, green and blue in each channel. Each color has a value from interval [0, 255]. OpenCV uses by default an inverse color sequence, i.e. BGR. That means a colorful depth-map will change when imported as shown in Figure 3.1.

While colorful images typically have three channels, grayscale has only one channel with value from [0, 255] and some colorful images might have a fourth channel for transparency. The grayscale can be read by explicitly specifying *cv2.IMREAD_GRAYSCALE* or integer *0*, e.g. *img = cv2.imread('img.jpg',0)*. [34] For problematic colorful image with transparency, it is a little complicated to convert back to RGB, because `cv2` ignore the transparency by default mode. [35] This transformation is written in Appendix B.3 *ground_truth_mask()*. The number of channels is critical, this part of script was written when a problem for reading from ground truth image occurs.

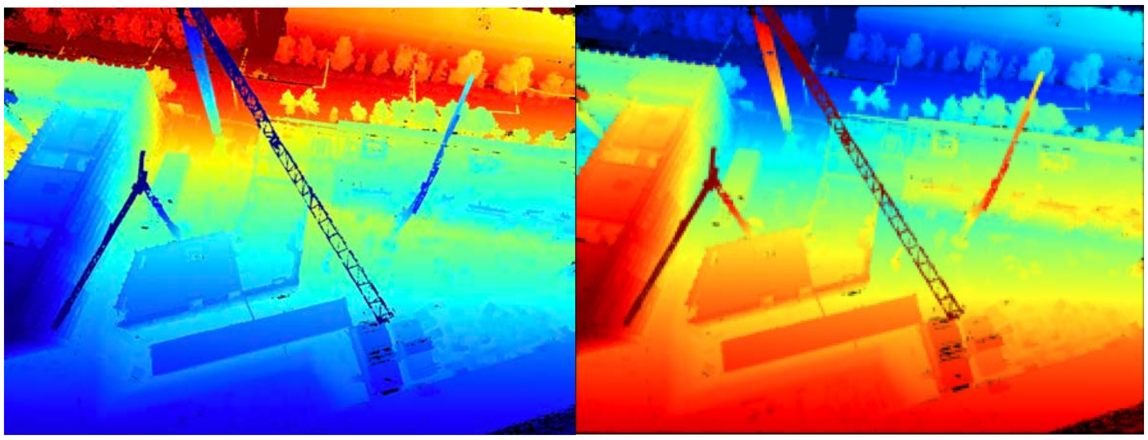


Figure 3.1: RGB image becomes BGR by default after imported via *cv2.imread('img.png')*

Except the numerical values, an image also provide some invisible information. Exchangeable Image File format (EXIF) is a standard that specifies the formats of images recorded by digital cameras, including smartphones. It provides static information of the camera and dynamic information such as orientation (rotation), aperture, shutter speed, focal length, metering mode, and ISO speed information, as well as current date and time. Focal length, camera direction and GPS can help determine the spatial relationship.

If we don't manually specify intrinsic parameters, they will be extracted from the embedded EXIF information. In case of construction site, the set of photos should normally be captured by the same physical camera, thus the intrinsic parameters can be shared between all images. To prevent ungraceful exit if this information varies in some images, it's less error-prone to choose the default parameters in Colmap.

### 3.3.2 Representative images and the Ground Truth

We have to think, which kind of object can be representative to evaluate. It has to be relatively easy to isolate from others, which means there should be an obvious difference between the desired object and background. And the object needs to be more complex than just white wall or gray floor, so that the different quality can be reflected by a visible variation on the object. According to these requirements above, the tower crane is a perfect choice. It is representative due to its very unique height compared with others and very complex structure. If the reconstruction can provide a better result on a tower crane, we have no reason not to believe that the reconstruction on other fine objects can also be improved. Besides, the crane towers have to keep static during the image capturing.

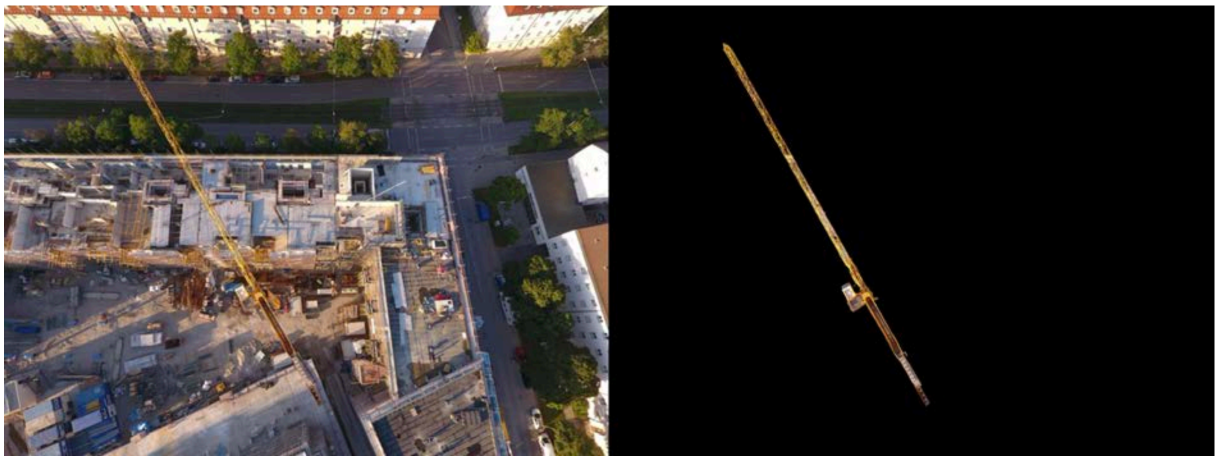


Figure 3.2: Undistorted image DJI_0038 and its corresponding labelling for tower crane (4013 x 3005)

We picked some images as the benchmark for evaluating the quality of depth-maps. These images should have tower crane located not too askew to the camera. In other words, the distances between each point on the tower crane and the camera position are close with each other, so that the crane can be easily isolated by assigning a small range of depth. Otherwise, more irrelevant objects would also be included in this range.

Photo editors, such as Photoshop, are used to pick out the tower crane manually. There are some points for attention. Firstly, the labelled image must be taken after undistortion stage. An undistorted image looks the same as its source image. But by checking its information in terminal with *file image_name*, we can see the difference on its resolution. In our case, the source image has resolution 4000 x 3000, while the undistorted one has 4013 x 3005. Even if the difference on resolution changes in 1 pixel, it leads to either incompatible operation or enormous error during the pixel-wise comparison. And the Labelled image needs to be as precise as possible, since it works as ground-truth, from which the automated depth-map deviates. The ground truth is the contour of the tower crane, everything not belongs to it has to be removed, including the interior in the grids. It is time-consuming to have a precise ground truth, so we have only the DJI_0038.JPG labelled. By zooming in, another harmful phenomenon named purple fringing was found, which will be further discussed in Chapter 5.1.

### 3.3.3 Extraction from Depth-Map

By running patch match, Colmap generates depth-maps in format compressed binary files, i.e. `.bin` files. These raw depth-maps cannot be opened directly, hence Colmap provides a very useful function *read_array(path)* in their GitHub repository under

\`scripts/python/read_write_dense.py\`. The output of this function is a 2-D matrix, each value of which represents a distance from the pixel to the camera.

There is one thing interesting — the value of this 2-D matrix is not the actual distance. There is no metric in the model. If we have one image in different data sets, e.g. a small data set with 42 images and a big data set with 420 images, the distance value in its depth-map can vary. But if everything is proportionally expanded, the relationship is still the same in the depth-map for this image. We can still extract the desired object by a bigger range, as long as the tower crane has a relative obvious difference on the distance with surroundings. More of this will be discussed in chapter 4.4.1.

### 3.3.3.1 Image Thresholding

On every image, we can use simple thresholding to isolate a certain range of depth for the object we need, e.g. tower crane. But this rarely returns a satisfied result on true-color images. Similar with temperature in thermography, a perfect depth-map provides colors only dependent on the spatial relationship. Through some tests, we also know that the absolute color value doesn't matter if it's HSV, RGB or even Grayscale.

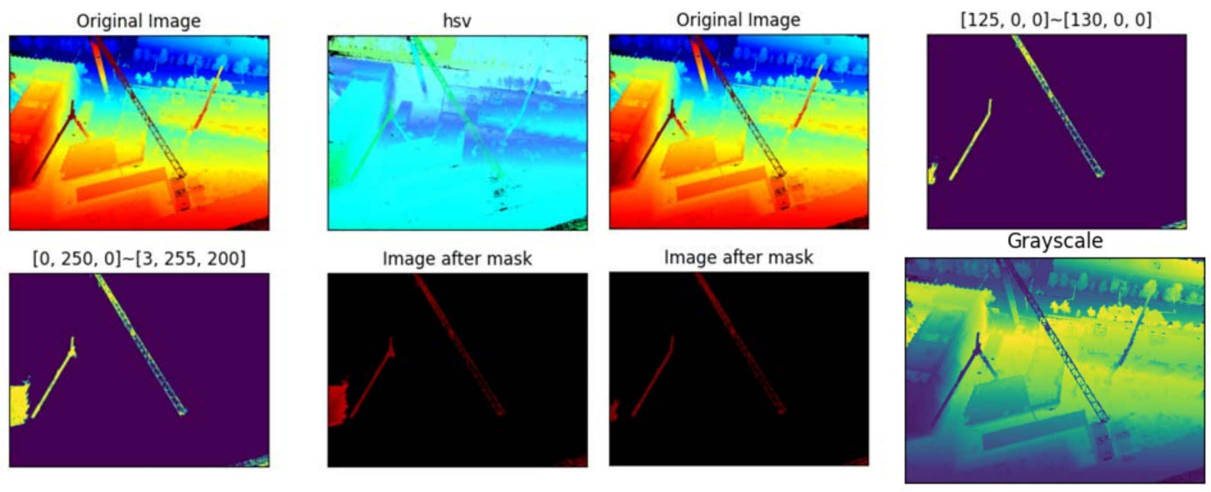


Figure 3.3: Thresholding on HSV and RGB image. Down-right is the corresponding grayscale.

As written in the beginning of this chapter 3.3.3, Colmap, or at least the function *read_array()*, produces depth-maps in single channel matrix. That turns to grayscale when using *matplotlib.image.imsave()* to output an image. Just like shown in Figure 3.3, it makes no difference when extract the contour of object from grayscale. It's even more simple to set the range. Later we also directly use depth-range to filter the object by function \`filter crane\` in Appendix B.2, the idea is congruent.

Here, the straight-forward function *cv2.threshold()* is used to get black-white images. In the sample code bellow, first argument is the source image, which should be a gray-scale image. Second argument is the threshold value which is used to classify the pixel values. If the pixel value is more than the threshold value, it will be assigned to 255, which represent white. And value 0 is pure black. The result mask would be an image with only 0 and 255. The last argument *cv2.THRESH_BINARY* represents the type of thresholding. [36]

```
ret, mask_img = cv2.threshold(img, 40, 255, cv2.THRESH_BINARY)
```

### 3.3.3.2 Region of Interest — Apply Stencils

Region of interest (ROI), in this case is a proposed region from the original image. Either by distance or color isolation, the object selection is purely based on the distance data, and the colors just display distance implicitly. This leads to an unavoidable problem, that sometimes there are more objects which have exact the same distance range as we search. Like shown in Figure 3.4, there some buildings locating in the same depth range with tower crane. These extra objects would reduce the precision of the detection, if we compare this mask with the mask from ground-truth. More about precision and recall will be discussed in chapter 3.3.4.

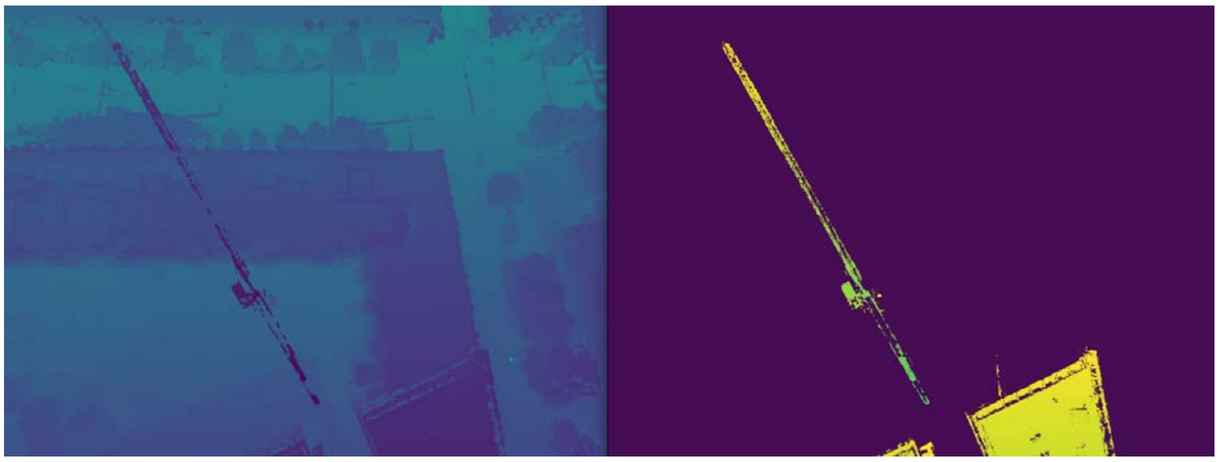


Figure 3.4: Original depth-map and its corresponding binary mask for the range of tower crane

To have an accurate evaluation on the desired object, the range of detection could be shrunk in advance. We call it stencil, which is generated by putting an enlarged mask from ground truth on the original depth-map. The area to detect is thus reduced to the size of the stencil. The interest pixels, namely what we are evaluating, locate mostly around the crane, which means they can be on the crane or inside the grids or outside near the crane. Any error in the reconstruction process may cause the deviation of pixel position. But the deviation will also not go too far away in most cases. Growing by a proper size in pixel, the mask should cover the most deviations around the objects and also avoid all other misleading extra objects like roofs.

We can see how good it works in figure 3.5, the roofs are successfully removed, and the crane finally looks very similar with the mask generated from ground truth, which is then convincing to evaluate the reconstruction quality by pixel-wise comparison.

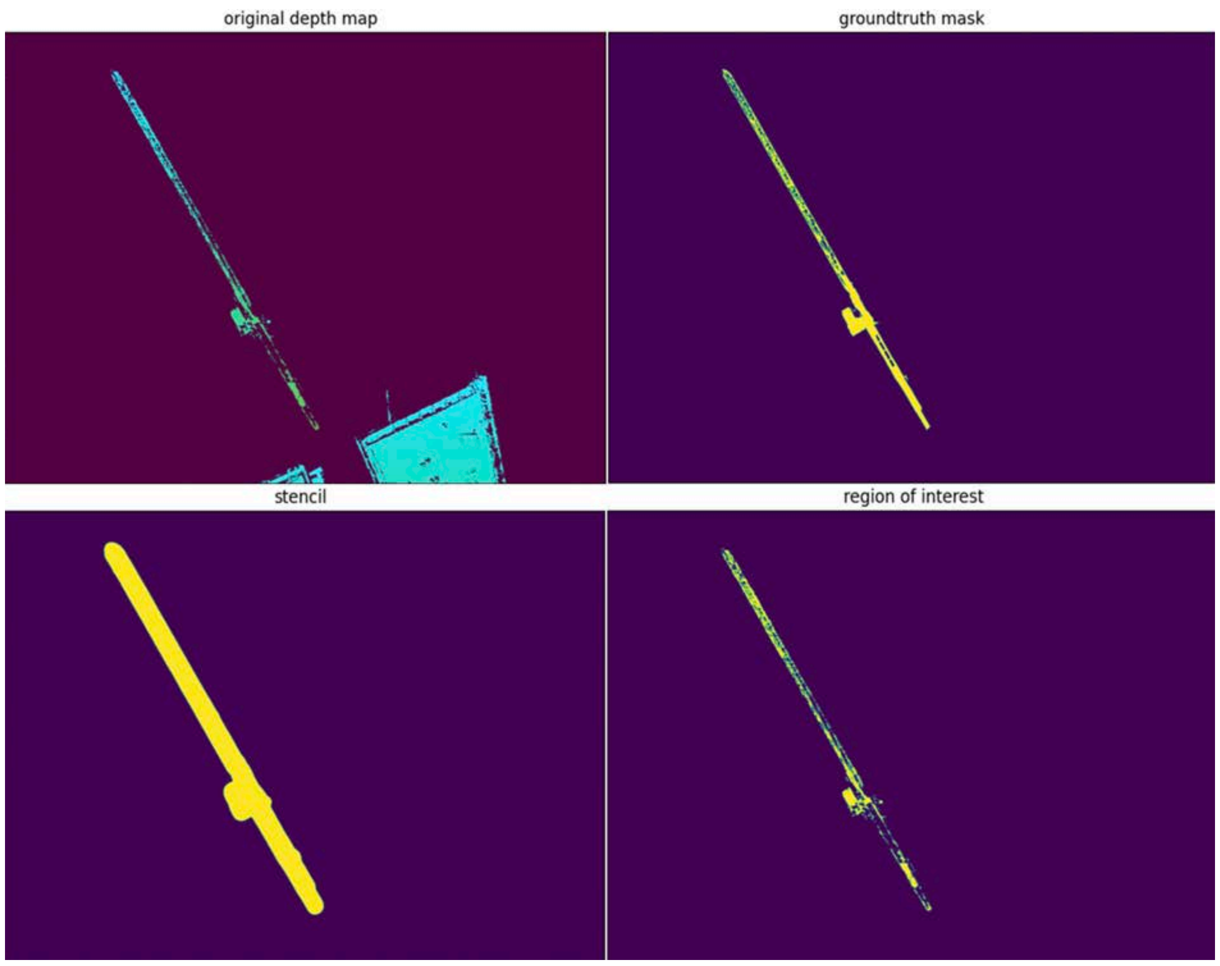


Figure 3.5: Mask from original depth-map (up-left), Ground truth binary mask (up-right), Stencil by Gaussian blur (bottom-left), Binary mask after applied stencil (bottom-right)

However, the drawback of this is also obvious, if there are some undesired objects very close to the observed object, but have the similar distance to the camera position, it can be hard to exclude. To reduce this, we shall select some images that don't have this problem to set up the evaluation depth-maps. In our case, as shown in figure 3.5, the tower crane in DJI_0038.JPG has a safe distance to surroundings.

#### 3.3.3.3 Object-Isolation in Depth-Map

There are two ways of selecting objects — by a range of color or distance.

Color isolation is not as precise as the original distance value. A small range of distance is assigned with a certain color. And that means, if the RGB number [127, 0, 0] means a range for 100~101 meters, but the objects in distance between 100.5 and 101 meters are what we need, it would give back more objects than expected by selecting the color.

Although distance value is more precise, selecting objects by a range of color is more straightforward. We could directly extract the RGB color value on the object with Photoshop or some similar software instead of looking into the large array of numbers, where we can hardly figure out the numbers representing the desired object.

There is another mixed way with help from *matplotlib.pyplot*. When depth-map is directly plotted by its depth value instead of gray or color channels, user can interactively see the depth range of crane by moving mouse cursor from its one side to the other. As absolute depth values vary at times, this mixed way turns to be more efficient.

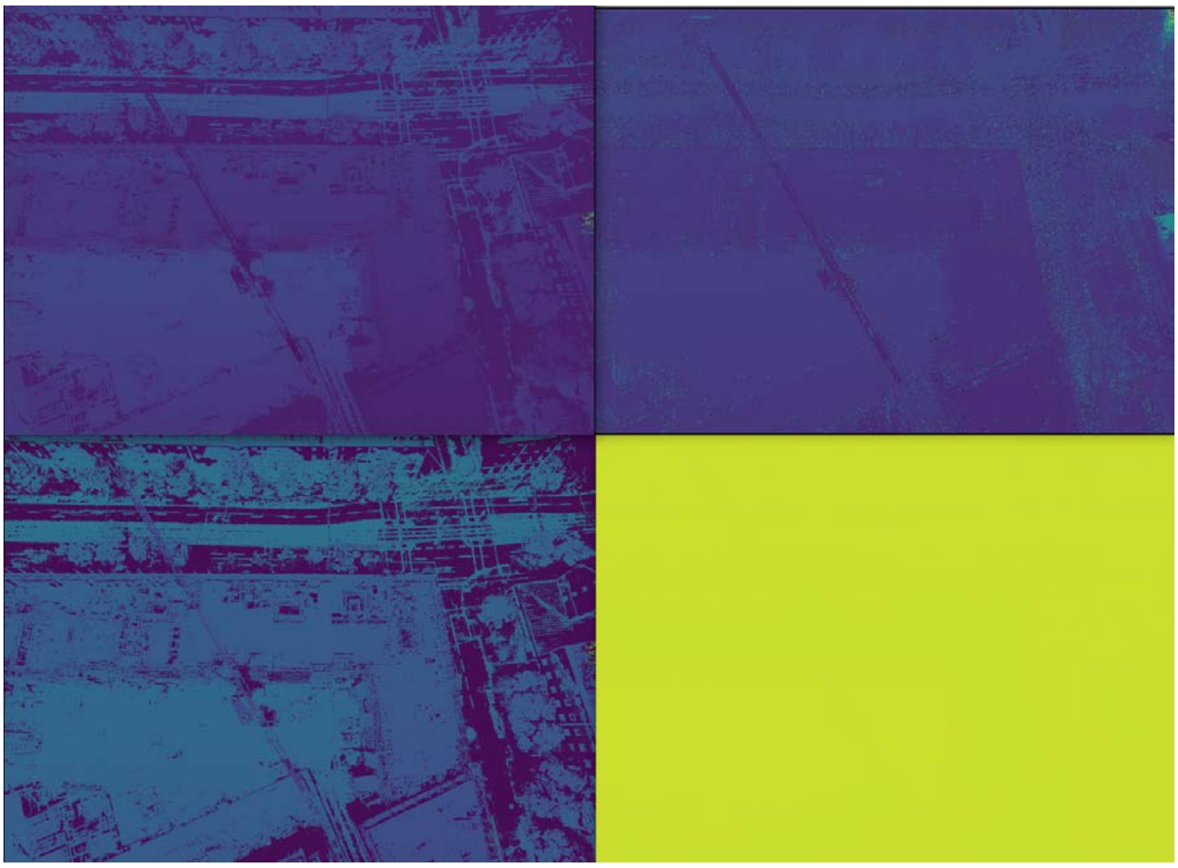


Figure 3.6: Geometric, outliers filtered (up-left). Photometric, outliers filtered (up-right), Geometric, outliers not-filtered (bottom-left), Photometric, outliers not-filtered (bottom-right)

No matter which method is used for finding the desired depth range, the outliers have to be considered. We can notice later from table 4-6, the minimum and maximum in a photometric depth-map are heavily deviated, which leads to too big range of depth values and thus bad visibility on the plotted image. As shown in figure 3.6, without running Appendix B.2 -> *filter_crane()* by eliminating the values which are bigger than 5 * mean or smaller than mean / 5 leads to invisible tower crane, especially for photometric depth-map. Although the image in bottom left looks not bad at first glance, the color on crane is still lighter.

### 3.3.4 Evaluation of Depth-Map

With the mask purely for tower crane, we can compare it with the mask generated from hand-labelled image (figure 3.2). The basic idea is:

real image -> hand labels -> mask A

depth-map -> color/distance isolation -> mask B

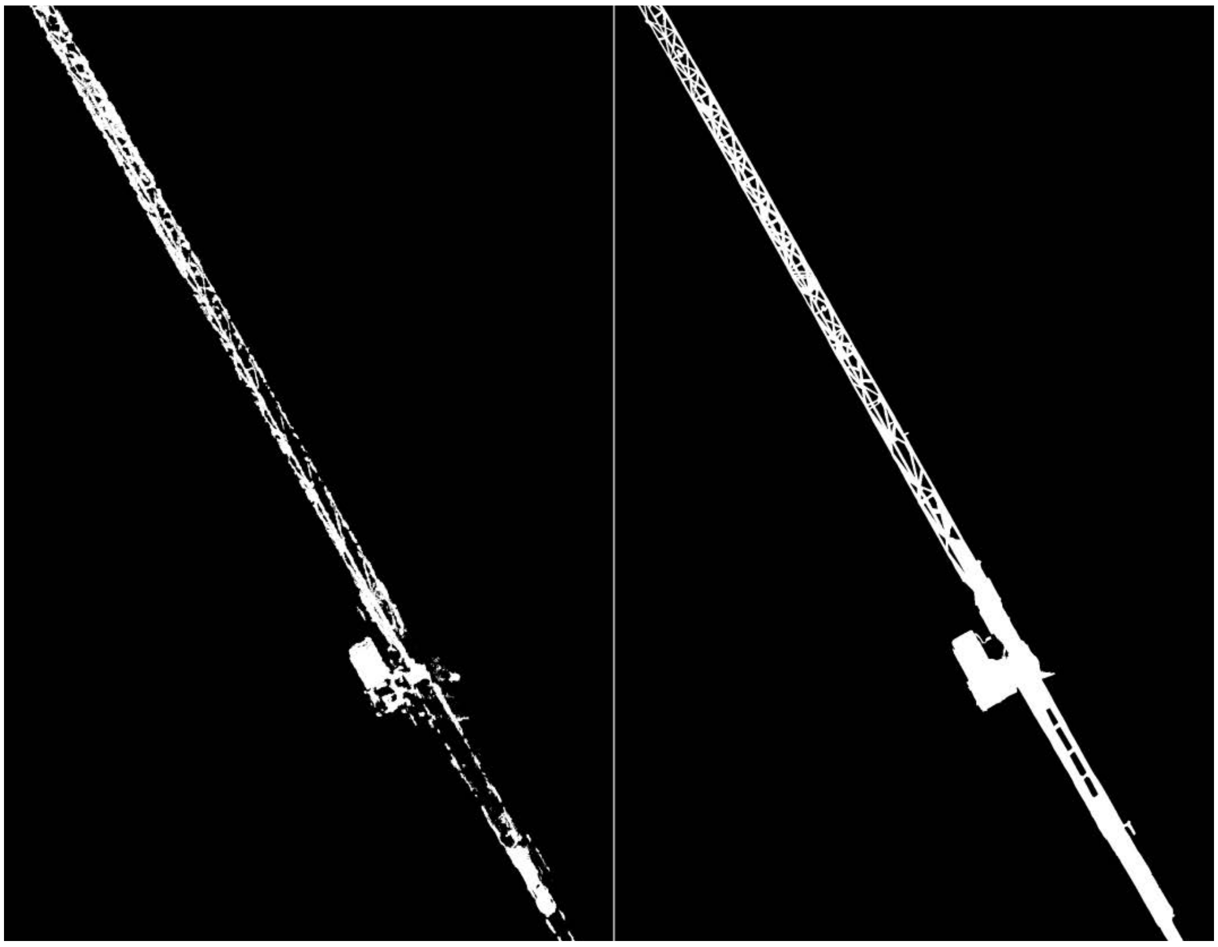


Figure 3.7: The tower crane on image DJI_0038.JPG to evaluate. Left is the isolation from depth-map, right is the hand-labelled contour as ground-truth.

We want to figure out not only that it is worse than ground-truth, but also how much it worse than. OpenCV provides the possibility of arithmetic operation on images [37]. Since these mask-images all have the same resolution, we can quantify the difference by subtraction. Minimizing the resulting error becomes the objective of optimization, because the quality of depth-map reflects the quality of the reconstruction. There is an interesting saying in Chinese, "pick a general from dwarves", which has a similar meaning with this situation.

The error of masks can be subdivided into difference, precision and recall. Generally, precision means how much among the selection is correct and recall means how much we have found from the whole correct set. An interesting example would be: considering you ask your dog to get all the oranges in the house for you. It brings 2 oranges and an apple back to you. You are not satisfied why it got only 2 oranges, but then you find out that there are indeed 2 oranges, no more. So, the precision is 2 oranges among three objects which is 66.7%, and the recall is 100% since your dog has already find all of the oranges.

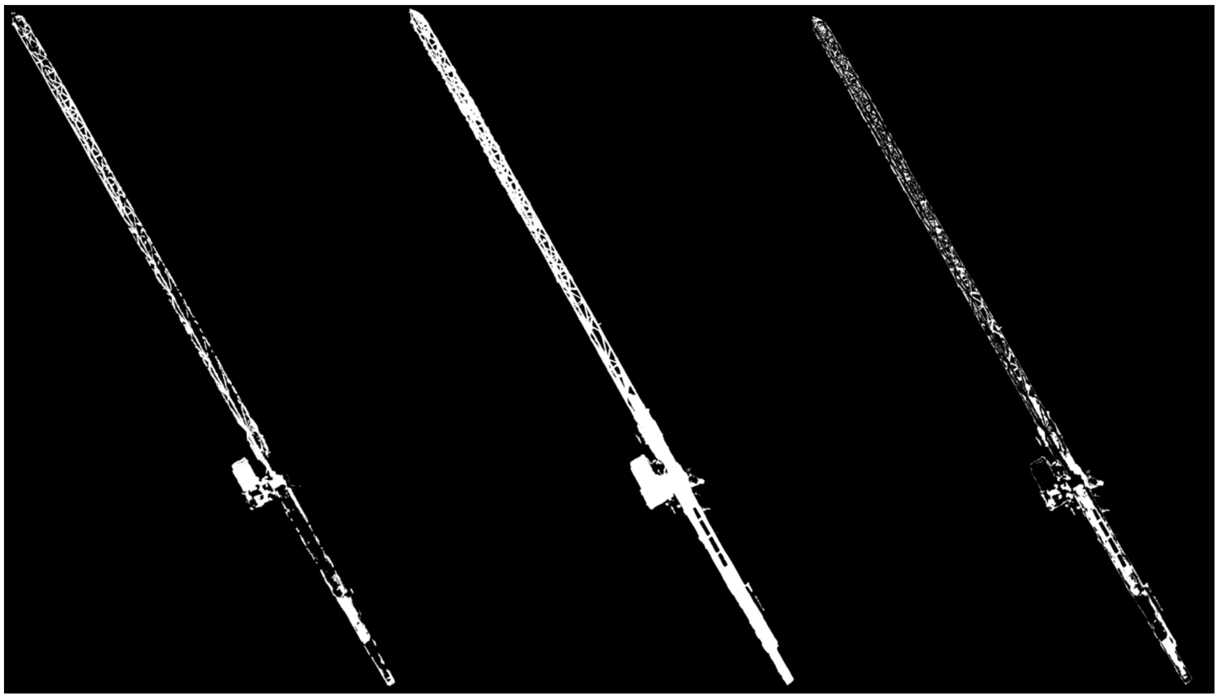


Figure 3.8: From left to right: Intersection, union and difference.

Back to our object detection, difference stands for all the pixels after absolute subtraction. The precision error stands for the white pixels existing in automated depth-map but are actually not real in ground-truth. The recall error is the white pixels not found in the ground-truth. Actually, difference equals to the sum of the precision and recall error.

Besides, we have to pay attention to the negative value in case 0 minus 255. The library function *cv2.absdiff()* thus used for calculating the difference, e.g:

*difference = cv2.absdiff(img_ground_truth, img_detection)*

Figure 3.8 provides three intermediate images — intersection, union and difference of the automated mask and ground-truth. We can get precision error by subtracting the ground-truth from the union and get recall error by subtracting the intersection from ground-truth. So, the optimal result would be an improvement on both precision and recall. For specific use-cases, we can also increase the weight of precision if surroundings are significant or increase the weight of recall for having a more complete structure of tower crane.

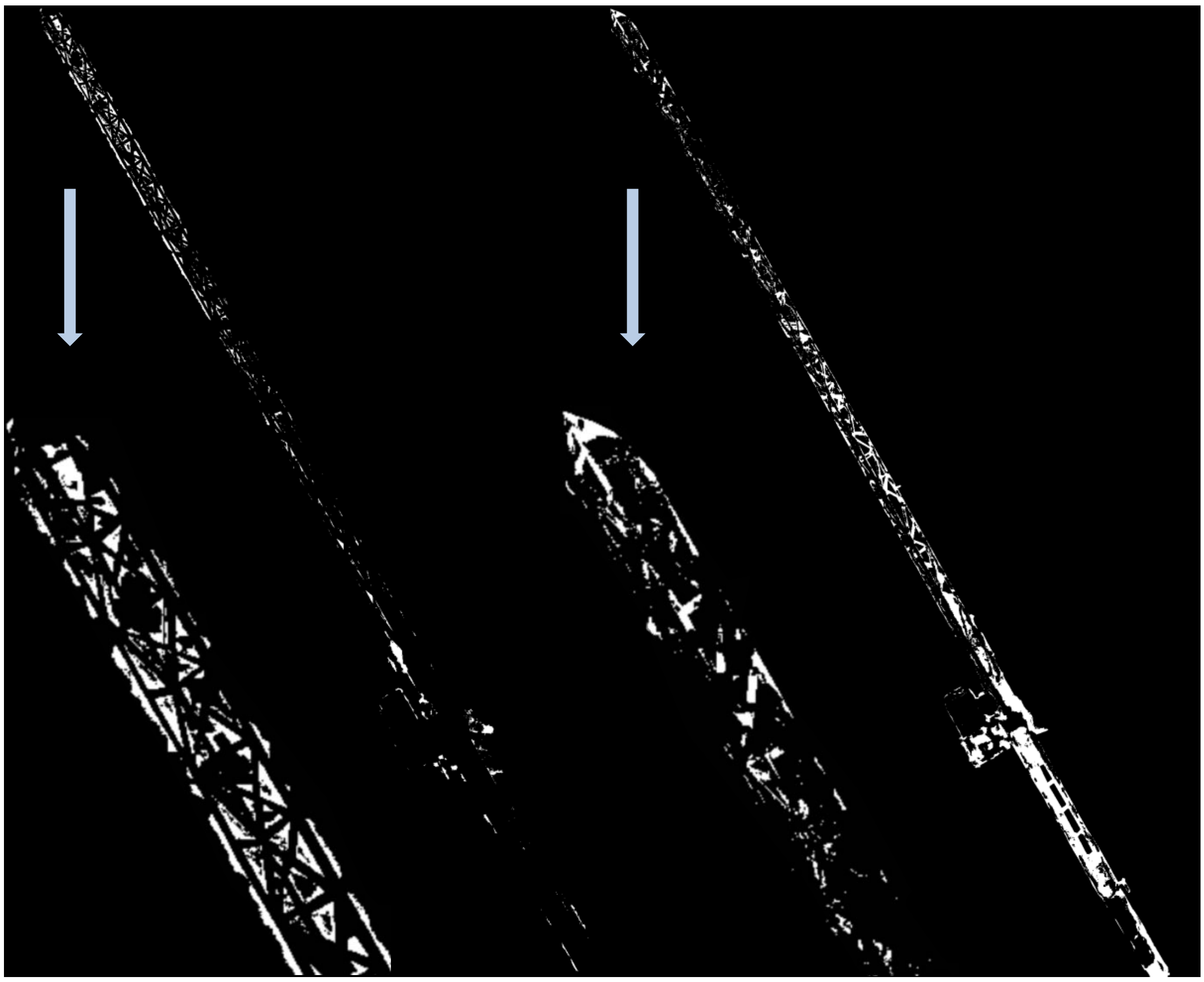


Figure 3.9: Precision error (left) and recall error (right).

In figure 3.9, we can see the contrast with regard to precision and error. Having a look at the magnified area, where a segment of typical fine structure is displayed, we can find the over-identified pixels on the left and the unrecognized pixels on the right. This imperfect depth-map shows the need for improvement. An ideal automated result should produce both images in figure 3.9 with white pixels as few as possible.

## 3.4 Optimization

Targeting a better result of the depth-map, an iterative optimization concept is here proposed. Due to different kinds of parameters and thus their plenty combinations, this optimization may generate a lot of computation for its wide range of possibilities but is theoretically feasible on the high-performance server. An optimal parameter combination could be found by automatically reconstructing and comparing with its previous results.

### 3.4.1 Optimization at Conceptual Stage

Subject to the changeable depth value in the model (Chapter 4.4.1), realizing a full-automated optimization is temporarily difficult. The concept of optimization here is thus restricted to the patch match stage, since the absolute depth value of feature is fixed after the SfM process.

However, the elapsed time for patch match is the most time-consuming process in Colmap, as later shown in Chapter 4.1. Even for the only patch match stage, the computation might require long time to find out an optimum. The number of iterations is hard to estimate and can easily exceed a hundred even if we modify just few parameters. And it is risky to use this unverified evaluation as the criterion. Only small portion of the results are to keep, because the computation produces a lot of intermediate data which need cleaning up regularly. We cannot know if it works until the optimization finishes.

To guarantee the controllability, in other words, to gain more knowledge on how the parameters affect the model than just numerical values of error and to correct the potential wrong direction in time, this design of optimization is discussed here limitedly for feasibility and serves as a good proposal for further study. Hence, a result of automatic optimization is not included in Chapter 4.

### 3.4.2 Local Search Method

It's difficult to announce that one of the parameter combinations is the best for reconstruction. The idea for optimization comes from the local search, which move from solution to solution in the search space by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. [38] In our case, instead of a time bound, we use a counter of failures when the result gets worse or out-of-boundary. If the *failures* occur for one parameter over certain times, this parameter should be dropped for further optimization. In cases for independent parameters, the comparison is over different values on a single parameter, with all the rest parameters keeping same with other models.

The greedy goal is to find a "global maximum", i.e. the depth-map with smallest error compared with the ground-truth. This can be formulated in terms of search space and target. Our search space can be generated based on an estimate range of parameters. As shown in Figure 3.10 [39], several situations are listed. Basically, this search would continue when it rises on the curve and terminate if the neighbors are worse. Since the path does not matter, the reconstruction could start from any point in its search path. With randomly restarting at some points, the cases like going down in a monotonically decreasing segment or ending at a bad local maximum can thus be avoided.

The optimization would stop when it got sufficient attempts in the search path, which means all the involved parameters have reached the limit of number of failures and removed from the parameter list successively. At last, the best parameter combination so far would be assumed as its global maximum.

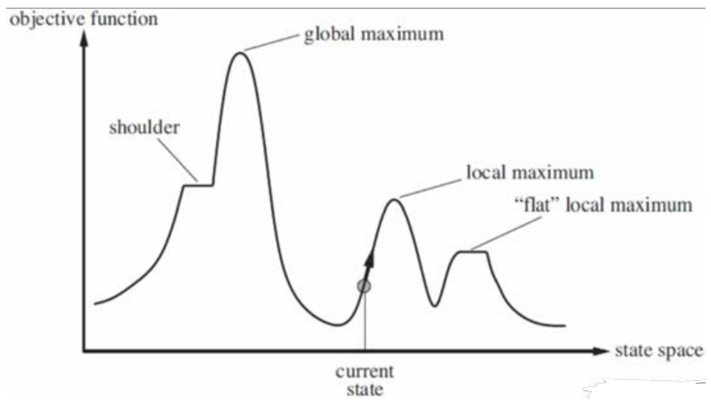


Figure 3.10: A qualitative search path of local search method.

### 3.4.3 Critical Parameters in Patch Match Stage

By consulting comments from the source code and documentation of Colmap, we can get a rough idea about which parameters would influence the reconstruction significantly and also know some linkages of parameters. The parameters that have a relatively big influence on the result should be marked as more significant and may be assigned with a larger tolerance of failures to be processed cautiously.

With the provided parser application, we're able to fully control the parameters in patchmatch stage from a corresponding text-file. Moreover, the parameters to modify iteratively can be substituted after dollar sign, e.g. *window_radius=$win_r*. And this parameter will be assigned by passing value in run-time, similarly as *workspace_path*.

Here are some meaningful parameters listed which are worth customizing or iterating.

Depth range is set to *-1.0f* by default. Although the absolute depth values would change out of control, a coarse range of depth for construction site doesn't change significantly in MVS and thus *depth_min* and *depth_max* can be provided to shorten the automatic inference and prevent exaggerated outliers.

Colmap suggests in their FAQ that for weakly textured surfaces a large patch window radius *--PatchMatchStereo.window_radius* and reduced filtering threshold for the photometric consistency cost *--PatchMatchStereo.filter_min_ncc*. [40] As our use case for construction site has many fine structures and partially meet the weakly textured surface, these two parameters are to be studied for an optimization. Additional to the quality, reducing the window radius can speed up the dense reconstruction. This contradictory effect requires a balance between them.

The photometric depth-map seems to have many outliers, and *filter* is used to filter out the excessive details. For tower cranes, fine structures usually don't have continuous surfaces, it can be harmful to filter out the jib and mast, which were found from the images "difficultly". We can thus compare a filter-true + geometric-false version with filter-false + geometric-true. The reason why we plan to try reconstruction without geometric consistency constraint is, turning off the geometric consistency check (*--PatchMatchStereo.geom_consistency false*) will speed up the computation. And if the result is passable, it might be worth a trade-off of efficiency against slight improvement to avoid geometric consistency check, not to mention a decline in the worst case.

To refine the filter, more parameters like minimum triangulation angle can be adjusted besides minimum NCC. And other parameters such as number of coordinate descent iterations can also vary in a certain range.

### 3.4.4 Iterative Method and Step Size

In order to execute the optimization automatically, the range of value and initial step size in advance. For each parameter, there is a failure counter — once beyond endurance, this parameter will be kicked out of the optimization process. And the value of this parameter in history that delivers the best result so far, will be taken from the list.

Hence, we need a dictionary to store the parameter-value pairs. More precisely, the "value" of this dictionary should be a list to hold values that are currently being adjusted. This kind of list is denoted in short by "optimizing list".

And we need another similar dictionary for storing the values that brings improving results. In other words, the values in this optimizing list should progressively increases the benefit to reconstruction in order. And the last value would always be the best for this parameter.

Here is an example in Python:

```
# These four values are: min, max, initial step size and failure counter

parameter_step = {'window_radius': [2, 10, 2, 0]}

parameters_dict = {}              # create an empty dictionary for storing history

best_parameters_dict = {}         # subset of history, progressively better values

parameters_dict['window_radius'] = []         # empty optimizing list

best_parameters_dict['window_radius'] = []    # empty optimizing list
```

The program should firstly run with a random value generated between the min and max for this parameter, e.g. *window_radius = 7*, This value will be recorded into the *parameters_dict[window_radius]* and *best_parameters_dict[window_radius]* at the same time. And the next run will use this parameter added a step size, i.e. *window_radius = 9* at this time. Assuming "9" delivers a better result than "7", this number "9" will then be stored into *best_parameters_dict*, which is [7, 9] now. It's clear that the last value in the *best_parameters_dict[window_radius][-1]* is the best value for *window_radius* so far. And since "9" is better than "7", we want to go further in this direction for further augmentation of the positive effect. In next run, the value would be 11, which exceeds the limit by *max = 10*. That is counted as a failure, and now we have to introduce another function *get_stepsize()* for cases that fail to deliver a better result.

As shown in Figure 3.10, sometimes it almost meets a crest of optimization, but the step size is just too big and exceeds the distance to the crest. It doesn't always mean that this value cannot go this way, sometimes it is the step size that matters. To avoid this, we should take care of "hill climbing" in that graph. If the next value shows a decline on result, or jumps out of scope like in this example, the step size should be carefully reduced. How much it needs to be reduced depends on the index of the best parameter *best_parameters_dict[window_radius][-1]* from tail of the *parameters_dict.* This index from tail is counted by *index_of_best_parameter - len(parameter_dict[])*. If this distance from tail is one, the default step size will be reduced to half, and for two it would be one-fourth.

Here is the corresponding code segment in Python:

```
# E.g., best_parameter is best_parameters_dict['window_radius']
# and parameter_history is parameters_dict['window_radius']
def get_step_size(best_parameter, parameter_):
    bpv = best_parameter[-1]     # best_parameter_value
    index_bpv = parameter_history.index(best_parameter_value)
    distance_from_tail = len(parameter_history) - index_bpv
    divisor = 2 ** distance_from_tail
    step_size = int(default_stepsize / divisor)
    return step_size
```

(This function *get_step_size()* should be further modified to handle "overloading" for general purpose, since the *step_size* for *window_radius* must be an integer, but some other parameters accept floating numbers.)

To pursue the global maximum, the start value of this parameter will be randomly assigned again, like *window_radius* starting from "3" and iterate with *step_size* further. At last, when the maximal number of failures is reached, this parameter is then removed from the optimizing list and the *best_parameter_dict* keeps it as a harvest.

More conceptual code is shown in Appendix B.5.

# 4 Experiments and Results

In this chapter, different test cases based on the former expounded methodology and their corresponding analyzation of depth-map are presented. We are going to execute the reconstruction both on the selected images and the whole dataset. Altering parameters which leads to significant improvement and possibly time-saving will be the goal of modification for the construction site.

## 4.1 Hardware Information

Since the subsequent efficiency and some restrictions are hardware-relevant, some of representative information are hereby listed briefly.

The author has a gaming laptop, which could represent the use-case on local computation to a certain extent. The server, from the chair of CMS, provides an efficient computation exemplar on high-performance workstation.

As shown in Table 4-1, with 256 GB RAM for example, we can assign for Colmap a larger *cache_size*, which keeps more bitmaps, depth-maps and normal-maps in memory. A higher value of *cache_size* leads to less disk access and faster computation, especially for a dense consistency graph. [41]

Table 4-1: Basic hardware information of laptop and server.

| | CPU | GPU | RAM | System |
|---|---|---|---|---|
| **Laptop** | Intel i7-9750, 6-core | RTX 2080, 8 GB | 16 GB | Linux x86_64 |
| **Server** | AMD Ryzen 3990X, 64-core | TITAN RTX, 24 GB | 256 GB | Linux x86_64 |

## 4.2 Effect of DSP and Elapsed Time

As introduced in chapter 2.2, DSP is mentioned to be an outperforming variation of SIFT. The switch of DSP is specified in feature extraction stage of Colmap workflow by *colmap feature_extraction --SiftExtraction.domain_size_pooling=true*. And the DSP-SIFT has further influence on its sequential computation afterwards. Here, we reconstruct with and without DSP respectively for a smaller dataset of 39 images with resolution 4000 x 3000. More information of this small dataset is explained in chapter 4.3.

### 4.2.1 Elapsed Time and Model Analyze

To compute successfully on laptop, some settings have to be adjusted for performance. Although the documentation of Colmap provides an approximate formula for calculating GPU usage: " 4 * num_matches * num_matches + 4 * num_matches * 256 ", which doesn't exceed the limit of RTX2080 yet. Several attempts found that on laptop it's sometimes not possible to run a feature extraction with full resolution for this dataset — the process was killed ungracefully without further information. As a result, the flag *--SiftExtraction.max_image_size* has to be reduced to half. Besides, to have a faster process on laptop, the rest parameters all stay defaults from Colmap, while the experiments on server use the parameters as shown in Appendix A.1.

Table 4-2: Elapsed time in SfM, on laptop with *max_image_size=2000*.

| Time [min] | Feature extraction | Exhaustive matching | Mapper | Bundle Adjustment |
|---|---|---|---|---|
| **DSP=true** | 1.056 | 0.187 | 1.982 | 0.114 |
| **DSP=false** | 0.074 | 0.160 | 1.724 | 0.081 |

As for elapsed time, there is no apparent difference except the feature extraction stage, where DSP significantly increases the computational workload. And the increasing workload restrict the possibility of using higher resolution on normal computers. Except the mentioned the significance of GPU for feature extraction in Colmap's documentation, by querying `htop` while running we can see a high CPU usage as well. The server overcomes the additional workload easily with its adequate multi-threads. By comparing the elapsed time for feature extraction on the server, it has a surprisingly similar time cost with DSP functioning compared to without. It seems that the resolution, number of features and octaves play a bigger role instead of DSP with regard to speed, when it comes to high-performance computation.

Table 4-3: Elapsed time in SfM due to DSP, on server with full resolution.

| Time [min] | extraction | matching | mapper | bundle adjustment | patch match | fusion |
|---|---|---|---|---|---|---|
| **DSP=true** | 1.233 | 0.685 | 6.273 | 0.238 | 25.996 | 4.835 |
| **DSP=false** | 1.056 | 0.695 | 6.093 | 0.375 | 25.969 | 4.759 |

The information of results, including mean reprojection error can be retrieved by querying *colmap model_analyzer --path ./to/sparse/path.* The reprojection error is a geometric error corresponding to the image distance between a projected point and a measured one. It is used to quantify how closely an estimate of a 3D point recreates the point's true projection. With DSP activated, the points and observations increased while the mean reprojection error also becomes bigger, which is out of expectation.

Table 4-4: Results from model_analyzer, on laptop with *max_image_size=2000*.

| | Points | Observations | Mean reprojection error |
|---|---|---|---|
| **DSP=true** | 63008 | 234584 | 0.770022px |
| **DSP=false** | 53830 | 199469 | 0.704108px |

Table 4-5: Results from model_analyzer, on server with full resolution.

| | Points | Observations | Mean reprojection error |
|---|---|---|---|
| **DSP=true** | 150365 | 557388 | 0.596832px |
| **DSP=false** | 148074 | 553246 | 0.558640px |

To conclude, it costs longer time and generates more features with DSP. Through this experiment, we can also see the time cost of each stage. When the volume of dataset increases to full captured images, it can take over a whole day on the server for only patch match stage. It becomes necessary to append a flag *-d* for run container in daemon to prevent accidentally shut-down of process, as described in Chapter 3.1.1.

## 4.2.2 Depth-Map and Point-Cloud

Table 4-6: The numerical values in depth-map. "range_l" and "range_r" stand for lower and upper bound of the depth value of tower crane for extraction.

| | Type | Mean | Max | Min | Range_l | Range_r |
|---|---|---|---|---|---|---|
| **DSP=false** | geometric | 3.82 | 27.22 | 0.0 | 3.22 | 4.26 |
| | photometric | 5.62 | 352.82 | -189.08 | 3.22 | 4.26 |
| **DSP=true** | geometric | 3.84 | 17.47 | 0.0 | 3.23 | 4.29 |
| | photometric | 5.64 | 172.03 | -1962.73 | 3.23 | 4.29 |

To provide more information than mean reprojection error, the server is used to generate complete dense models for DSP and non-DSP version with all the rest variables controlled to be same. However, as the smaller dataset varies from the original dataset in SfM steps, the undistorted image DJI_0038.JPG has a different resolution with from ground-truth. The resolution of this image is 4013 x 3004 and 4012 x 3004 respectively in dense model with DSP-true and DSP-false. Thus, a straight-forward subtraction on images and compare their errors in precision and recall as in following analyzation is not feasible. Besides, to apply stencil to the depth-maps is also not available. Here, we can only have a coarse comparison on the depth-maps and point-clouds.

Executing geometric consistency check provides a much cleaner result, which provides a more apparent and less oscillated image comparison with ground-truth. In other comparisons afterwards, we will also mainly focus on the geometric version to evaluate the improvement or decline on the depth-map's quality. There's one thing to note, it doesn't mean yet that geometric is better than photometric version. The point-cloud generated from photometric version doesn't have apparently worse recovery on structures compared to geometric version. In fact, if one geometric version depth-map is outperforming, it's convincing that the corresponding photometric version outperforms others as well. At that point, the difference between photometric and geometric can be further discussed.

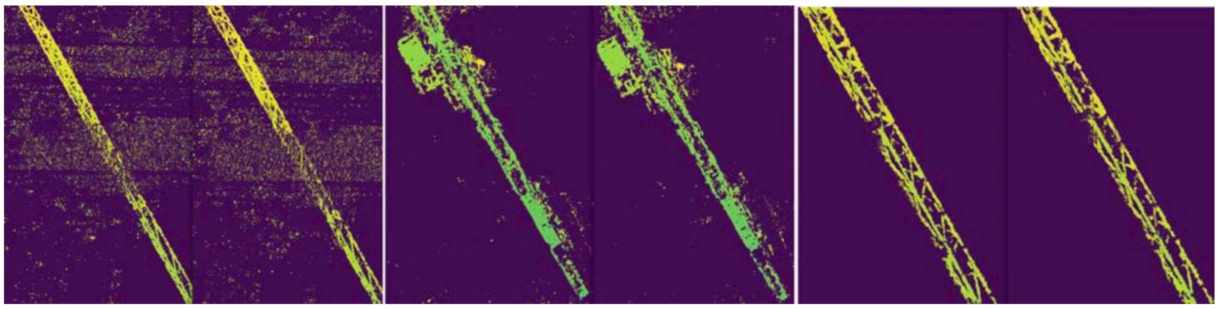


Figure 4.1: Three pairs of comparison of depth-maps with DSP-true(left) and DSP-false(right). First two pairs are photometric version and the last pair is geometric.

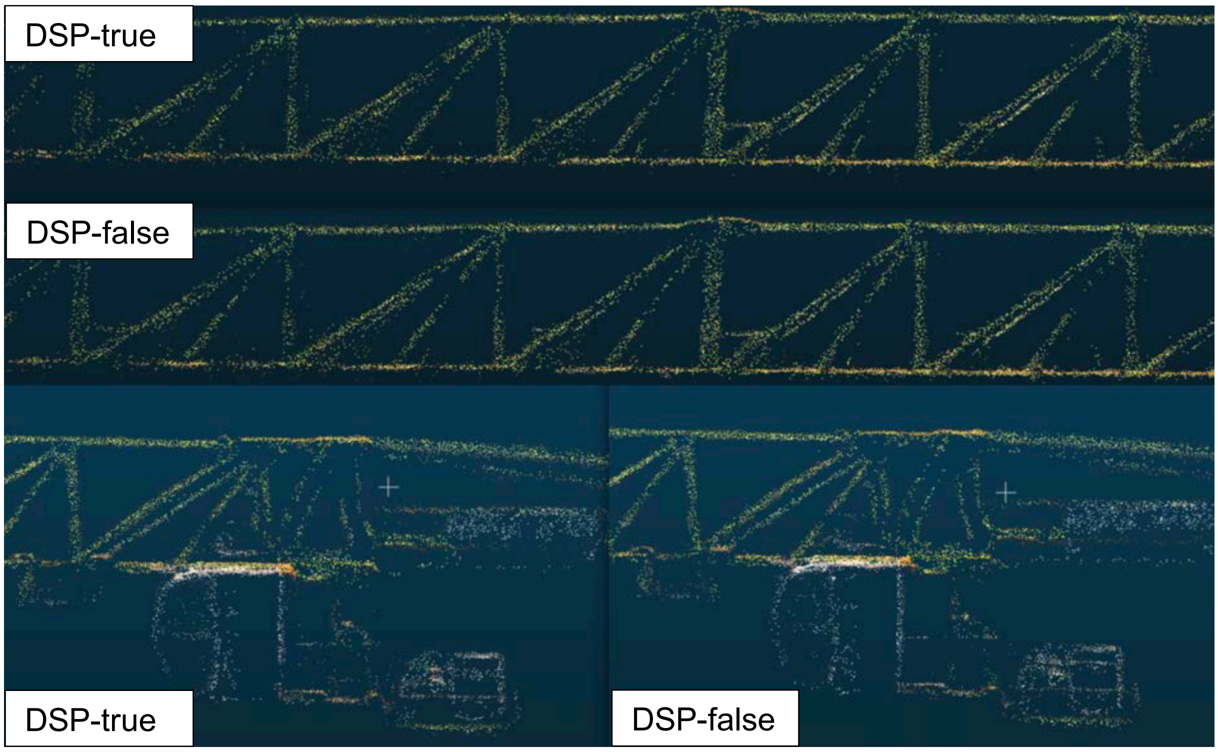


Figure 4.2: Two pairs of comparison of point-clouds generated with geometric consistency check.

With Table 4-6, it can be inferred that DSP mode rejects some outliers in "max" but brings extra outliers to the "min" in photometric depth-maps. And the geometric depth-maps filters all negative values, the DSP version has thus a more concentrated range of depth values. However, looking into the comparison in both Figure 4.1 and Figure 4.2, there are only negligible improvements by DSP. A more precise conclusion cannot be provided.

Because the DSP doesn't significantly increase overhead, in this thesis the model of complete dataset in the following is reconstructed with DSP activated.

## 4.3 Speed-up: Selection on Relevant Images

As demonstrated in chapter 4.2.1, reconstruction is very time-consuming. Rather than half hour, our original dataset has 484 images in total, which takes around 15 hours for the server to compute one piece of patch match stereo with geometric consistency check under default settings. And meanwhile, it also generates over hundred gigabytes on hard disk space.

Compressing images' resolution seems to be a way to accelerate, but this operation can be harmful for fine features. Moreover, the ground truth (i.e. the labelled image) is fixed on resolution and thus the stencil generated from ground truth is also with full resolution. If the resolution changes in patch match stage, e.g. with flag *--max_image_size 2000*, the pixel-wise image comparison would be very problematic due to different sizes of image.

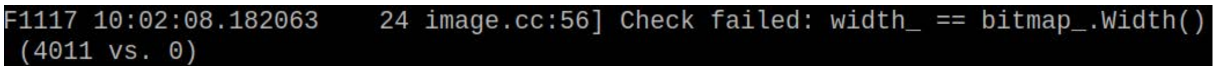
### 4.3.1 Overlapping Images via Colmap GUI

Since not all images are relevant to the representative depth-map, reducing the number of images involved in patch match stage should be possible. After querying the "Overlapping images" for relevance in "Database management" of Colmap GUI, the "image_id" of all the matched images are known. For image `DJI_0038.JPG`, there are 38 matched images. It means basically that only these 38 images are relevant to produce a depth-map pair `DJI_0038.JPG.*.bin`, which is corresponding to the ground-truth. Including the most important image `DJI_0038.JPG` itself, it's 39 images in total.

### 4.3.2 Modify the CFG File to Alter the Model

It would be very expensive to let Colmap do another computation from the very beginning for a smaller dataset. Not only will it take long time for SfM, but we will also have to label new undistorted image, although it varies very little, e.g. from 4013 x 3005 to 4017 x 3006 by test. Besides, the whole coordination changes and the absolute value of depth also varies a lot, which requires a new depth range to isolate the tower crane.

Instead of generating a totally new model with these 39 images, we choose to shrink the evaluation scope beginning from dense model. As patch match has the `work-space` in `$DATASET_PATH/dense`, the images to alter are restricted in the already undistorted images. The first naïve try is to directly delete the irrelevant undistorted images in `/dense` directory. It was unsuccessful due to the reason that Colmap still try to search the correspondence from the new images following its old model relationship. As shown in Figure 4.3, the error comes when it tries to work for an image, who is already deleted.



```
F1117 10:02:08.182063    24 image.cc:56] Check failed: width_ == bitmap_.Width()
 (4011 vs. 0)
```

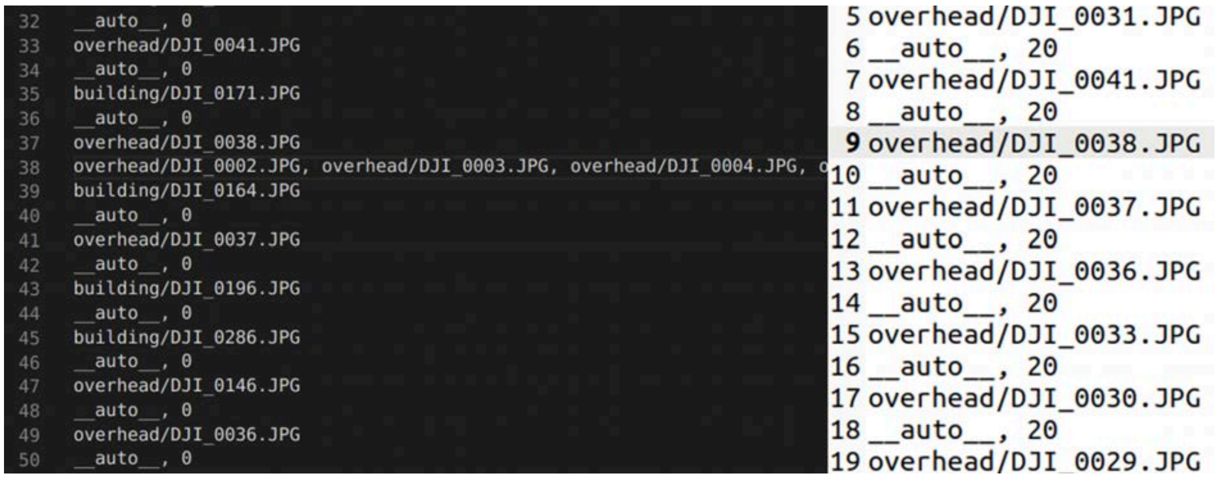
Figure 4.3: 4011 stands for the length of required undistorted image and 0 stands for the image which no longer exists

To overcome this problem, Colmap provides a way for manual specification of source images during dense reconstruction [42]. Under `dense/stereo/` there is a `patch-match.cfg` file, where all source images to be used are listed. In our case, it has 472 images by default, slightly reduced from 484 images. To generate a new configuration file for the reduced image set, a small script is needed instead of typing it manually. The source code is shown in Appendix B.6. Besides, this modification is later also used for comparing cases with different number of automatically involved source images.

There is one thing to note, we cannot generate only one single depth-map. Each two lines in the configuration file indicate an image to reconstruct and its relating partners. E.g.:

```
overhead/DJI_0038.JPG      # the first line indicates the depth-map we want
overhead/DJI_0024.JPG, overhead/DJI_0042.JPG, ...  # second line: partners
```

Although only `DJI_0038.JPG` is needed, we should keep not only this image with its partners, but keep this image and also its relating images with their partners! Otherwise, when Colmap get another image ID from partner list, it would go to analyze this image. And error occurs when this image doesn't show up as the "first line" with its partners.



```
__auto__, 0
overhead/DJI_0041.JPG
__auto__, 0
building/DJI_0171.JPG
__auto__, 0
overhead/DJI_0038.JPG
overhead/DJI_0002.JPG, overhead/DJI_0003.JPG, overhead/DJI_0004.JPG,
building/DJI_0164.JPG
__auto__, 0
overhead/DJI_0037.JPG
__auto__, 0
building/DJI_0196.JPG
__auto__, 0
building/DJI_0286.JPG
__auto__, 0
overhead/DJI_0146.JPG
__auto__, 0
overhead/DJI_0036.JPG
__auto__, 0
```

```
overhead/DJI_0031.JPG
__auto__, 20
overhead/DJI_0041.JPG
__auto__, 20
overhead/DJI_0038.JPG
__auto__, 20
overhead/DJI_0037.JPG
__auto__, 20
overhead/DJI_0036.JPG
__auto__, 20
overhead/DJI_0033.JPG
__auto__, 20
overhead/DJI_0030.JPG
__auto__, 20
overhead/DJI_0029.JPG
```

Figure 4.4: Wrong configuration with 38 images in next line on the left and the correction on the right.

Additionally, modifying the patch-match.cfg file directly is not workable due to the lack of permission — docker has root privilege, the old patch-match.cfg shall be replaced inside a docker container with instruction `mv new_patch_match.cfg /path/to/dense/stereo/patch-match.cfg`.

In this way, the workload of patch match is significantly reduced. The undistorted images stay unmodified under the dense directory but only a subset of them will be used for further reconstruction. Colmap follows the new configuration file to generate for the reduced subset both photometric and geometric depth-maps as well as the subsequent point-cloud. The speed of computation is thus accelerated.

## 4.4 Patch Match and Dense Point-Cloud

Patch match is a critical stage to improve the quality of point cloud. As proposed, depth-map acts as the evaluating medium. In this Chapter, some parameters in patch match stage are to alter, and we compare the difference of dense models by analyzing both depth-maps and point-clouds.

### 4.4.1 Changeable Depth Value

This problematic phenomenon has to be mentioned before we look into the results of depth-map. Any change in SfM part would generate different sparse model and thus leads to a different coordinate system of point-cloud, i.e. changeable absolute values in depth-map.

Each point-cloud has its own coordination, even two models only differ in very few parameters in SfM stage. Because Colmap infers a relative scale from images by a certain combination of parameters in SfM, and it varies more when the input or parameters changes more significantly. In chapter 4.2, the depth range of tower crane in DSP version, which was read manually with *matplotlib.pyplot* in Table 4-6, is slightly different from non-DSP version. It was not an inadvertent error but an indeed deviation of position. Using CloudCompare to have both point-cloud opened, the ghosting of tower crane can be seen in Figure 4.5 apparently.

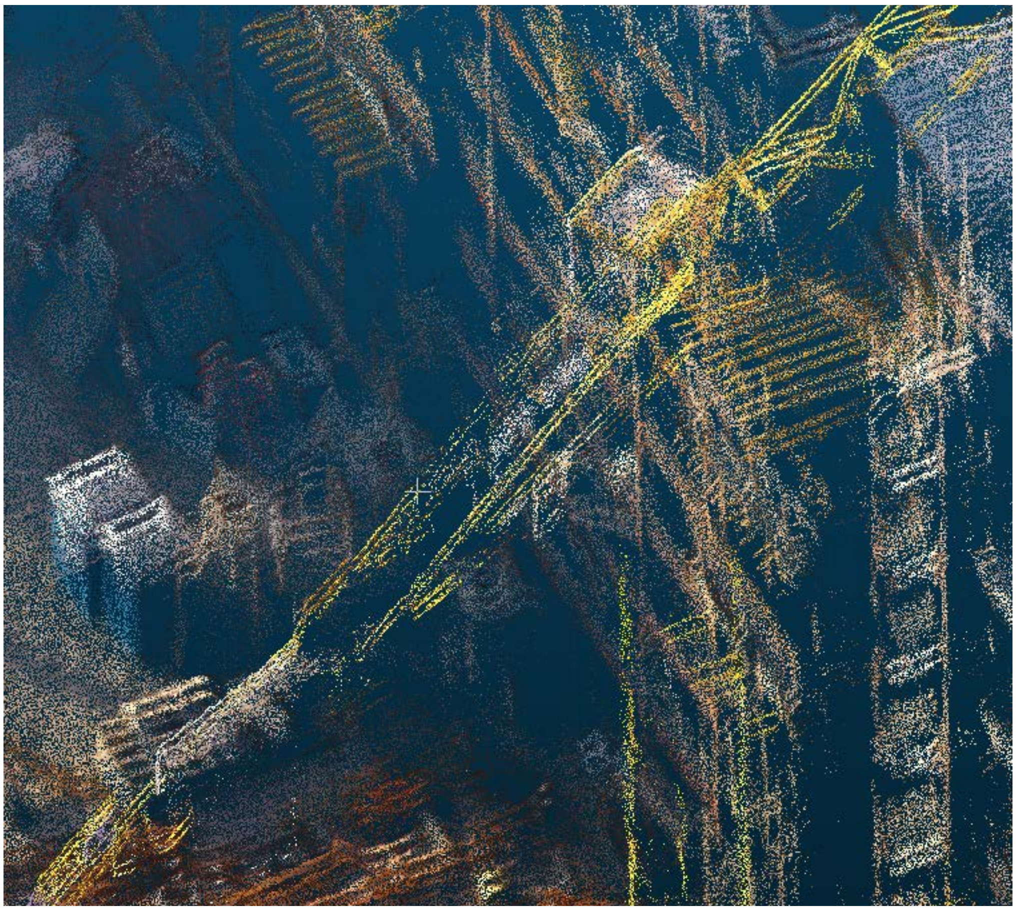


Figure 4.5: Two point-clouds in DSP and non-DSP version differ slightly in position or maybe also scale. All other parameters remain the same.

What's worse, this variation would interrupt a potential automatic processing. If the tower crane is used to be a medium for evaluation, it has to be detected and isolated in each execution, in order to compare its corresponding binary mask with the one from ground-truth. We thought for one scenario or at least for one dataset, there would be a fixed depth range for tower crane. But it's impossible to know the exact depth value or how it varies when some parameters changes. That means this range of depth can only be given in advance if the sparse model is fixed. For specific object to be the criterion, the isolation has to be done manually on depth-map so far.

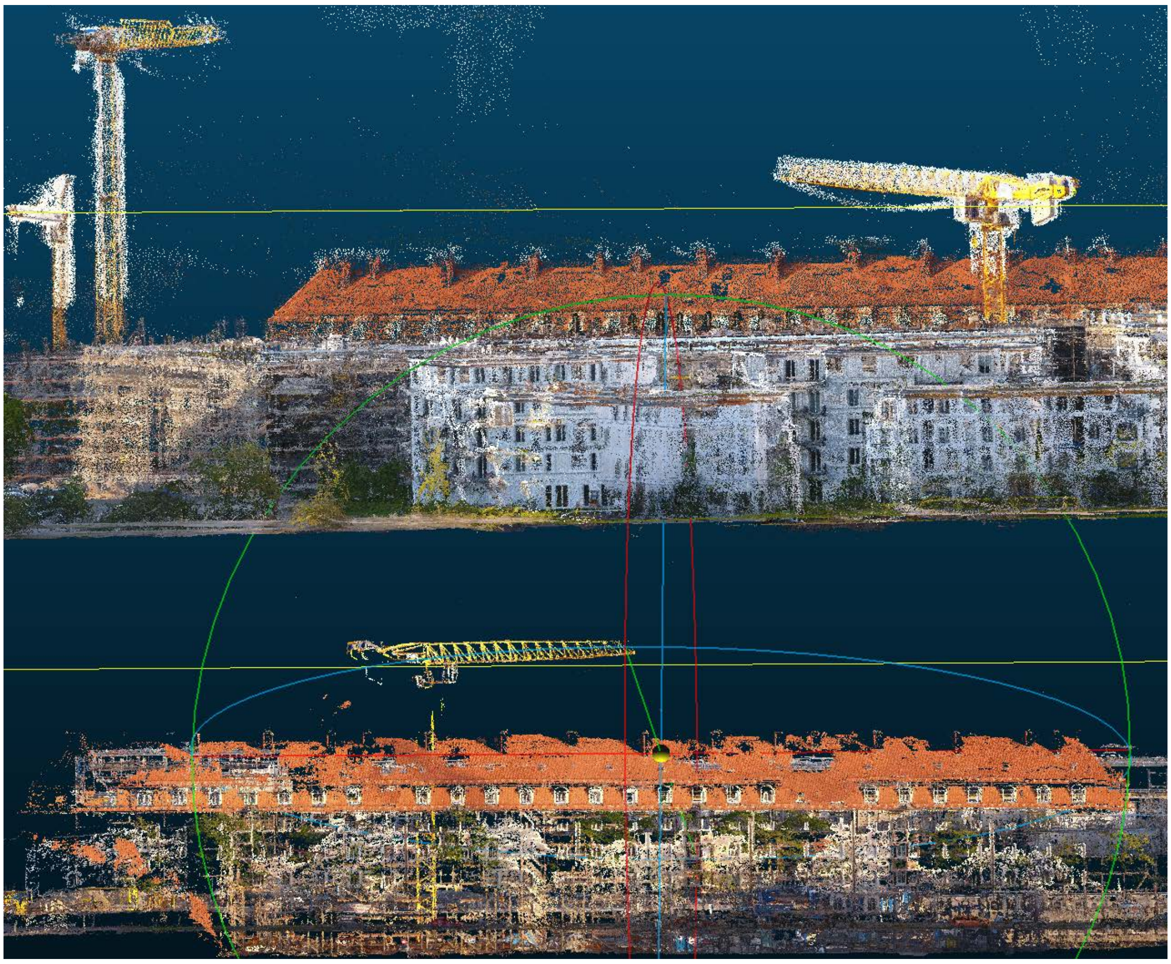


Figure 4.6: Point-clouds from automatic reconstruction (up) and DSP version (down) locate different and have different directions in one coordinate system.

Another example in Figure 4.6 above, two point-clouds are opened into CloudCompare. The automatic reconstruction is for the whole dataset with default settings, while the other uses the 39 images with customized parameters. As a result, they have totally different positions in point-cloud as well as the different depth ranges of crane tower in corresponding depth-maps.

We assume that the metric of coordinate system is stable after sparse model reconstruction, and the depth range for tower crane is steady [**4.66, 6.19**].

### 4.4.2 Number of Involved Overlapping Images

In Chapter 4.3, we know that Colmap has a configuration file `patch-match.cfg` for defining which images are involved in the dense model. And there is a second line after each image name, e.g. `__auto__, 20`, which means how many images (or explicitly which images) will involve with this image in the most overlapping sequence automatically. How this number influences the quality of reconstruction is in this chapter studied.

The 39 images we select are all of the relevant images with regard to `DJI_0038.JPG`. As a result, if the `__auto__, num` is set to `__all__`, it should have a same effect as `__auto__, 38`. We set the number of involved overlapping images to 5, 10, 20, 30, all, and the results are listed in the following tables.

For names in all tables below, "auto" means the number of automatically chosen overlapping source images, "time" indicates the computing time for patch match step in minutes, and "win_r" for *window_radius* in patch match step.

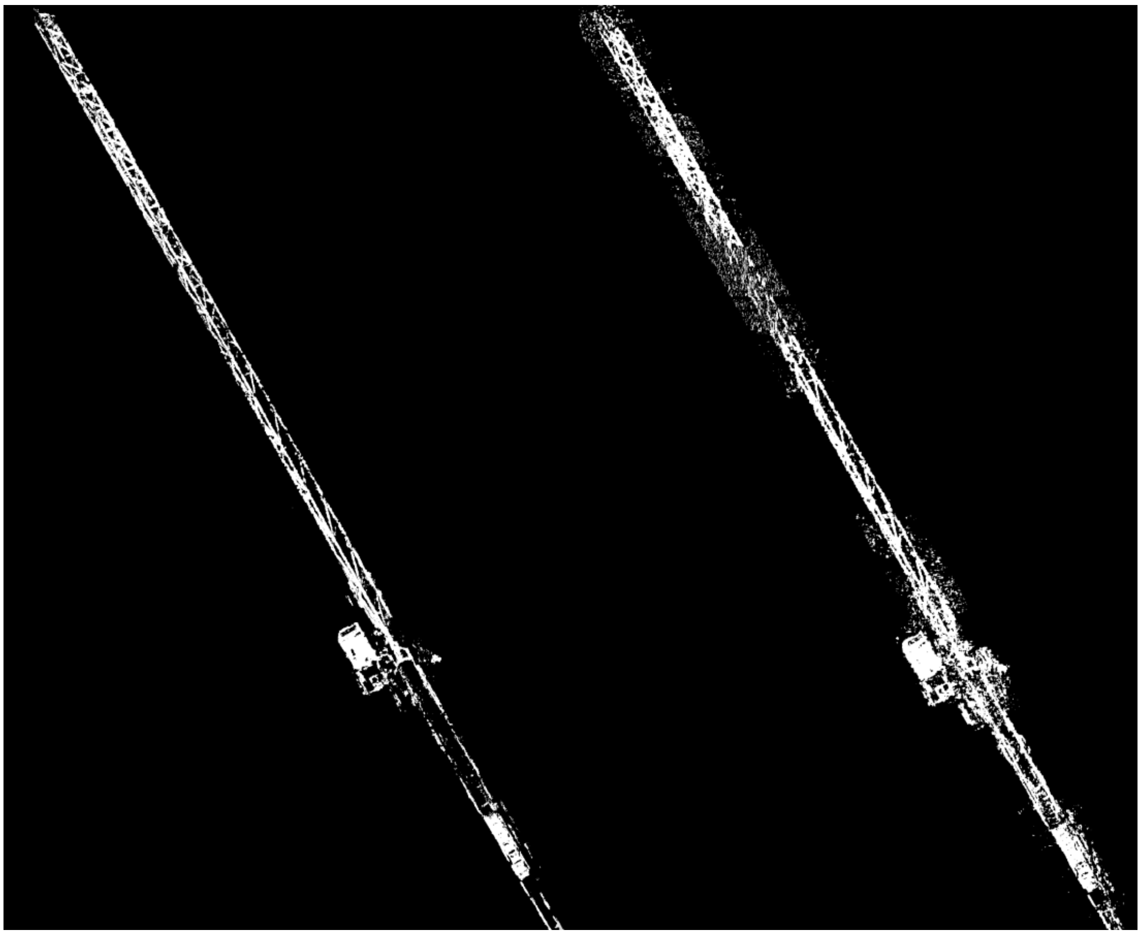
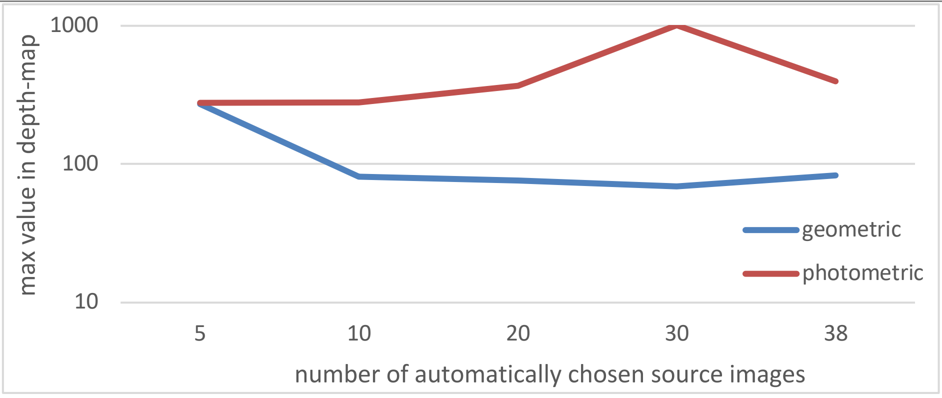


Figure 4.7: Geometric(left) and photometric(right) depth-map after stencil for "auto20"

Under the condition of only one sparse model, the depth coordinate system is supposed to be the same, which is corroborated by a steady depth range of tower crane. However, when we look into the Table 4-7 and its corresponding chart, the maximum and minimum depth values in photometric depth-map become larger surprisingly when the number of involved images increases. While the values in geometric depth-map are more concentrated by using more source images, which meets our expectation. Combining with Figure 4.7, we could infer that increased source images bring more features that include more outliers but also provide corrections on most points. Thus the depth-map filtered by geometric consistency check, gained the benefits and get smaller maximum. This phenomenon is shown also in the following semi-logarithmic line chart.

Table 4-7: The depth value results in depth-maps with different number of automatically chosen source images as variable. window_radius = 2

| Auto | Time [min] | Type | Mean | Max | Min |
|---|---|---|---|---|---|
| 5 | 73.1 | geometric | 4.75 | 272.07 | 0.0 |
| | | photometric | 8.09 | 277.48 | -360.04 |
| 10 | 38.9 | geometric | 5.24 | 80.88 | 0.0 |
| | | photometric | 8.14 | 280.43 | -260.48 |
| 20 | 26.0 | geometric | 5.56 | 76.58 | 0.0 |
| | | photometric | 8.14 | 370.22 | -456.08 |
| 30 | 69.5 | geometric | 5.60 | 69.09 | 0.0 |
| | | photometric | 8.14 | 1009.98 | -3717.64 |
| all (38) | 96.8 | geometric | 5.60 | 83.24 | 0.0 |
| | | photometric | 8.16 | 397.46 | -129.7 |



For reason that the photometric depth-maps have too many outliers which leads to a heavier oscillation with regard to the number of pixels in image subtraction, we choose to only compare the results of geometric depth-maps in Table 4-6.

This experiment shows a best result for number of automatically involved overlapping source image equals 20. It has less recall error meaning the area for tower crane is more complete. And it also has a not too bad precision. Additionally, in this case it costs apparently less time, since the information from overlapping is adequate but not redundant.

Table 4-8: Evaluation of depth-maps with different number of automatically chosen source images as variable. win_r = 2

| Auto | Precision error | Recall error | Difference | Type |
|---|---|---|---|---|
| 5 | 9529 | 63072 | 72601 | geometric |
| 10 | 9968 | 60089 | 70057 | geometric |
| **20** | **9875** | **59380** | **69255** | **geometric** |
| 30 | 9440 | 61455 | 70895 | geometric |
| all (38) | 9405 | 62552 | 71957 | geometric |

By comparing the errors and also the shape of tower crane in different depth-maps, it can be inferred that the Colmap becomes more prudent for display a pixel for on the crane when more images getting involved. There could be some conflicts when redundant images provide different information, so that the computation takes longer time and more pixels especially on the edge of structure would be rejected. That's why we choose `__auto__, 20`, a moderate number of images to avoid over-filtering.

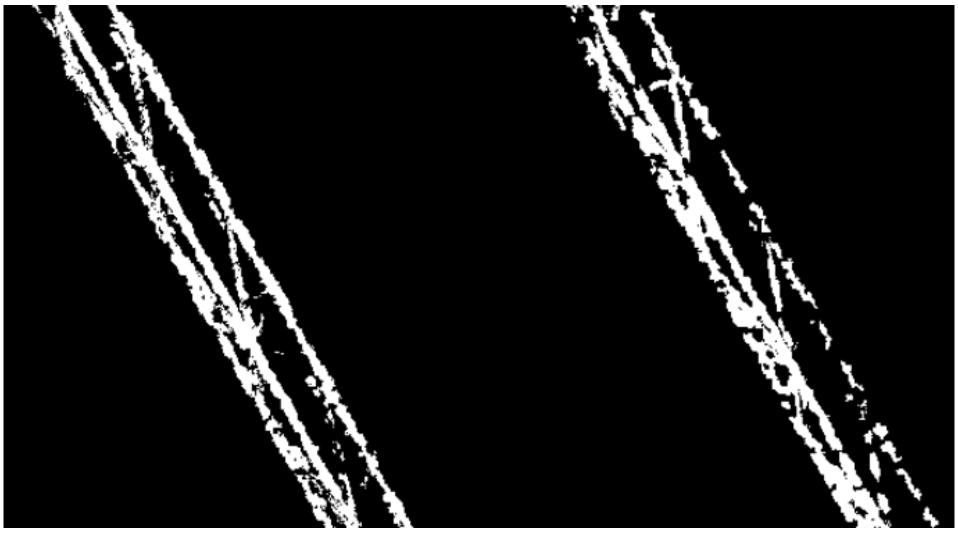


Figure 4.8: Left is by `__auto__, 5` and right is by `__all__`, having more overlapping source images surprisingly harms the contour of tower crane at times.

### 4.4.3 Photometric vs Geometric Consistency Check

We showed some results about photometric depth-map like in Figure 4.1. Most of time, a comparison of photometric version depth-maps is omitted. Compared with geometric depth-maps, they are more complete but with much more noise on non-crane points, which leads to heavier oscillation by counting pixels. Geometric depth-map acts as a filtered version that more possible noise dots are removed, where the minimal depth value is always perfectly zero, but the crane tower looks more heavily hollowed out. By directly comparing photometric version of depth-map with ground-truth, we should focus more on the recall, because the outliers in stencil significantly increase the error concerning precision.

With filter and without geometric consistency check, the process of patch match should save much time since the patch match is to proceed for each image only once. Hence, we test a filtered photometric patch-match without geometric consistency check and compare it with a not-filtered geometric patch-match.

Here, some comparisons of these two kinds of depth-map and their corresponding fused point-cloud are list to have a coarse evaluation. As we can see in in Table 4-9 and 4-10, the filter reduces the precision error from 35171 to 29272, but enlarge the recall error from 39407 to 43313. As reminded before, although it seems to have a not-bad improvement to the precision, some of these corrected errors may just from the non-crane pixels in stencil. But the recall is indeed reduced. The filter doesn't have a significant improvement on precision compared with not-filtered geometric version. Although geometric version have even a little lower recall than filtered version, it has much better precision which is worthwhile. If we look into the depth-maps in Figure 4.9, it's obvious that the filter did a bad job. Most details in that enlarged segment, which could be extracted nicely by geometric consistency check, are destroyed. In the original image, we can understand that this segment looks very confusing for the filter, since the background has a similar structure as crane.

Table 4-9: Depth values of filtered photometric version, not-filtered photometric version and not-filtered geometric version. (window_radius = 2)

| filter | geo_check | Time [min] | Type | Mean | Max | Min |
|---|---|---|---|---|---|---|
| true | false | 17.75 | photometric | 7.04 | 370.22 | -3.43 |
| false | true | 32.57 | photometric | 8.14 | 370.22 | -456.08 |
| | | | geometric | 8.07 | 2770.09 | -446.79 |

Table 4-10: Depth-maps evaluation of filtered photometric version, not-filtered photometric version and not-filtered geometric version. (window_radius = 2)

| filter | geo_check | Type | Precision error | Recall error | Difference |
|---|---|---|---|---|---|
| true | false | photometric | 29272 | 43313 | 72585 |
| false | true | photometric | 35171 | 39407 | 74578 |
| | | geometric | 23434 | 45140 | 68547 |

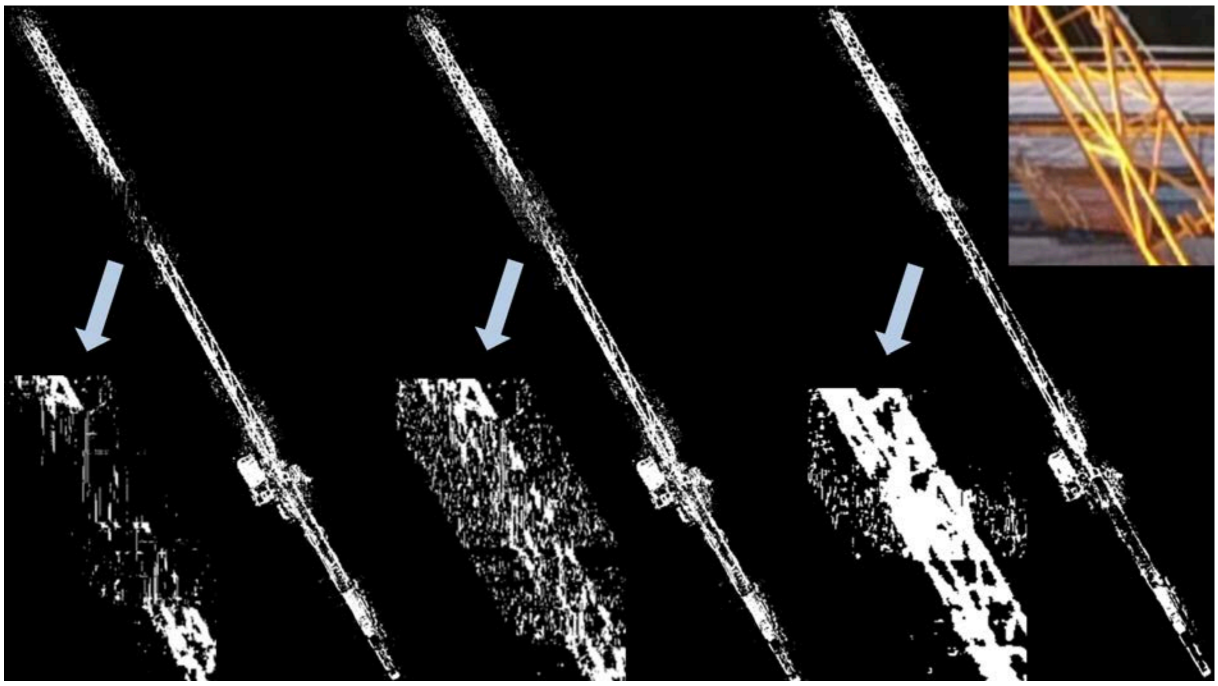


Figure 4.9: From left to right: filtered non-geometric version, not-filtered photometric version and not-filtered geometric version. This problematic segment is extracted from original image in color.

A ranking list of screenshots for point-cloud is shown in Figure 4.10. Although there are a lot of noisy dots in each photometric depth-map, the resulting point-cloud looks similarly clean as geometric version. From bottom to top, the filtered photometric version doesn't deliver a much different result with the not-filtered photometric version unexpectedly. But the geometric version provides a significant improvement on the point-cloud. Inferring the reason behind, geometric consistency check has filtered many outliers more effectively and precisely than filtering in fusion stereo stage, so that the rest structures are well-preserved by generating the dense point-cloud. Opposite of this, photometric depth-maps look more complete, but some of the structures are then eliminated together with outliers ungracefully in stereo fusion stage.

Additionally, it's interesting that either photometric or geometric version has some points that the other doesn't have. Having these two point-clouds opened as overlapping, the model is apparently enhanced. If there could be a way to merge photometric and geometric version, a better point-cloud may be achieved.

To conclude, not-filtered patch match with geometric consistency check delivers an optimal result in this case but takes longer time due to additional output for geometric depth-maps.

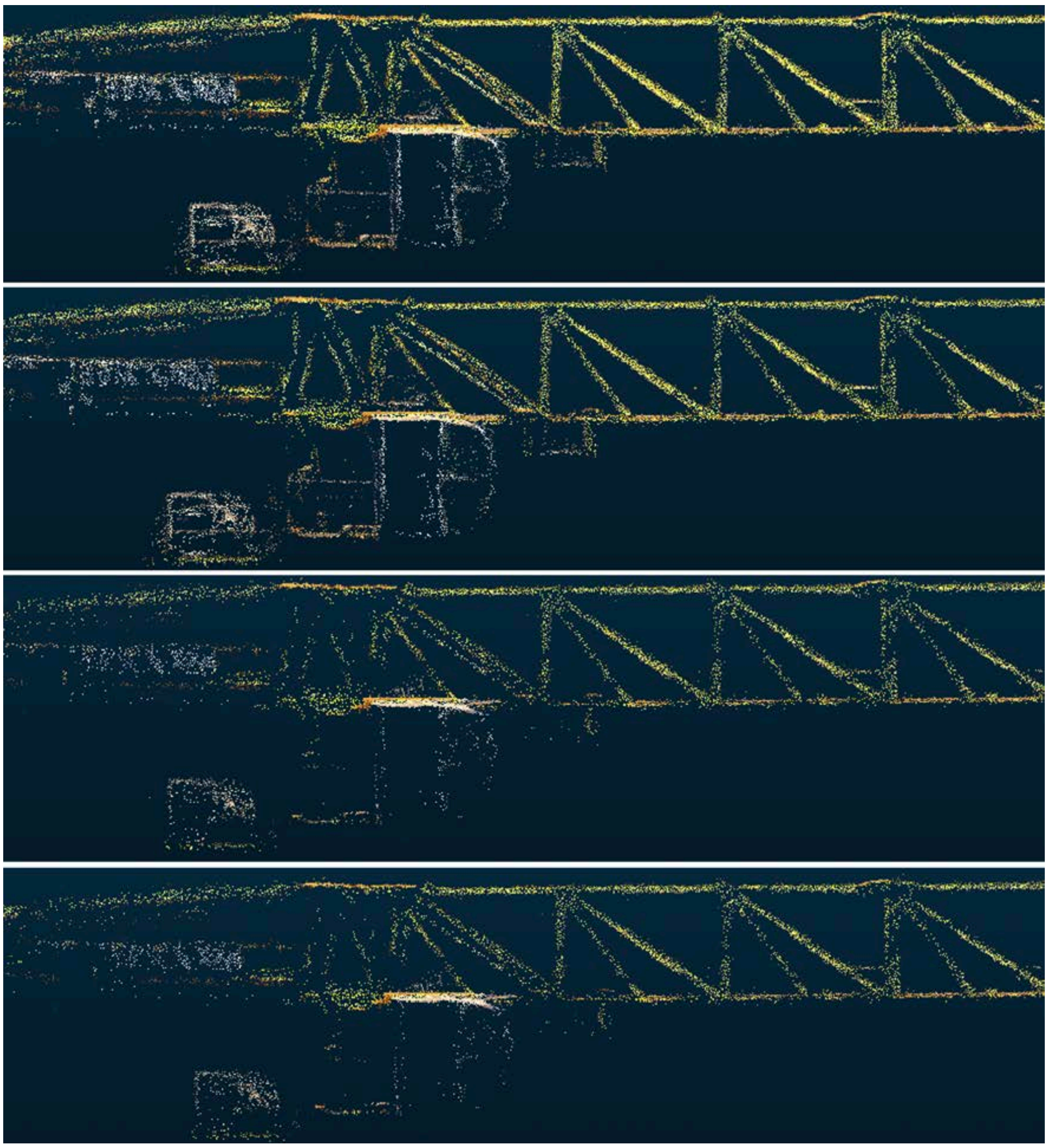


Figure 4.10: From top to bottom: superposed geo- and photometric without filter, geometric without filter, photometric without filter, photometric with filter

### 4.4.4 Window Radius and Window Step

Window radius is to measure the size of a patch concerning how many surrounding pixels should contribute to the reconstruction around a focusing pixel. Similar but not fully like repairing a stain in Photoshop, where the main content in the "window" will be magnified and some tiny isolated pixels will be removed. In Colmap, the tiny elements around a feature would appear in more windows if the size of window radius grows. Apparently, for bigger window radius, it costs much longer time since more pixels need processing. However, this "extra" work for fine structures like scaffolding and tower

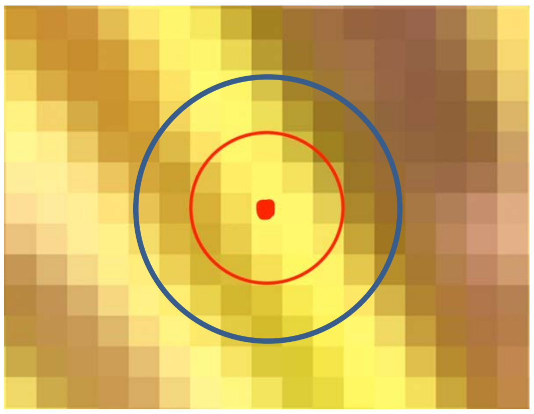


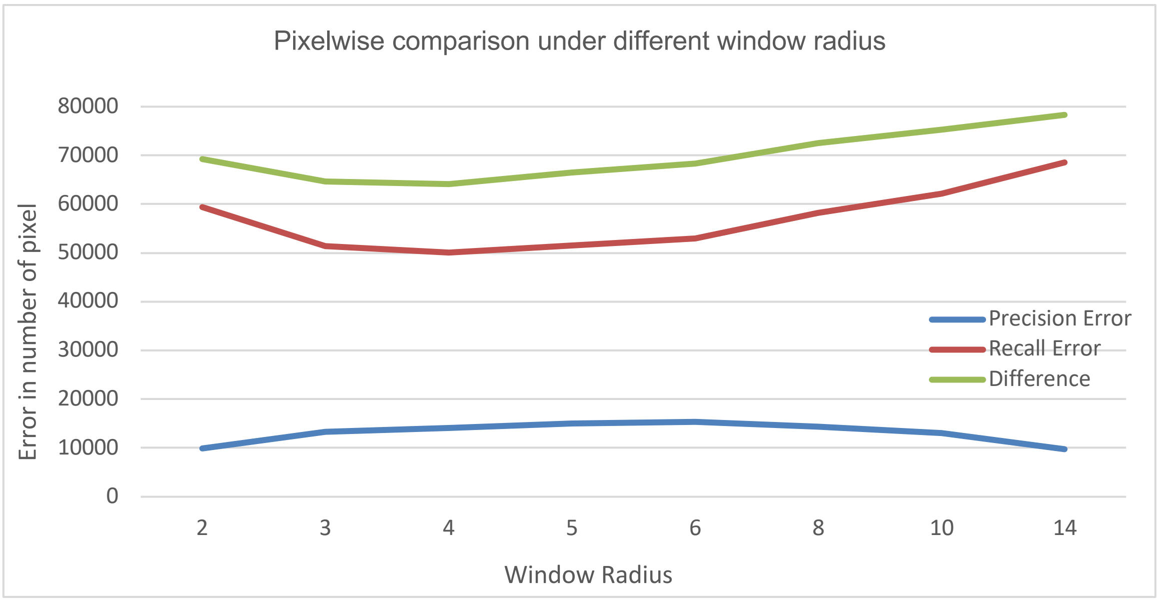
Figure 4.11: Schematic window radius.

crane might be harmful at times, since unlike texture-less surfaces the fine structure in this study only occupy very few pixels in width. A reconstruction with big window radius can be over-estimated by bringing too many textures.

To display the effect of window radius, we choose the values 2, 3, 4, 5, 6, 8, 10, 14 and keep other parameters unchanged. Bigger the window radius, more pixels get involved into the computation for one point. Actually, we could guess in advance that 14 is inappropriate for window radius, since it would be extremely time-consuming and obviously too wide for fine structures. But this bigger radius can bring out a more conspicuous contrast with cases having small window radius.

Table 4-11: The numerical values on depth-map and evaluation by pixelwise comparison.

| Win_r | Time [min] | Type | Mean | Max | Min | Precision | Recall | Difference |
|---|---|---|---|---|---|---|---|---|
| 2 | 26.0 | geo | 5.56 | 76.58 | 0.0 | 9875 | 59380 | 69255 |
| | | pho | 8.14 | 370.22 | -456.08 | | | |
| 3 | 34.3 | geo | 6.32 | 192.25 | 0.0 | 13271 | 51422 | 64693 |
| | | pho | 7.99 | 728.95 | -428.21 | | | |
| 4 | 210.2 | geo | 6.71 | 4755.69 | 0.0 | 14045 | 50071 | 64116 |
| | | pho | 7.96 | 841.52 | -444.33 | | | |
| 5 | 242.2 | geo | 6.94 | 44.31 | 0.0 | 14972 | 51536 | 66508 |
| | | pho | 7.91 | 274.82 | -447.72 | | | |
| 6 | 251.2 | geo | 7.08 | 23.45 | 0.0 | 15364 | 53028 | 68392 |
| | | pho | 7.93 | 457.79 | -89.47 | | | |
| 8 | 301.6 | geo | 7.23 | 702.86 | 0.0 | 14315 | 58242 | 72557 |
| | | pho | 7.92 | 709.17 | -36.27 | | | |
| 10 | 634.3 | geo | 7.30 | 23.44 | 0.0 | 13090 | 62201 | 75291 |
| | | pho | 8.00 | 4897.12 | -38.92 | | | |
| 14 | 991.6 | geo | 7.38 | 3665.25 | 0.0 | 9754 | 68567 | 78321 |
| | | pho | 8.33 | 23827.68 | -4817.85 | | | |



The results shown in Table 4-11 and its corresponding chart matches our assumption of optimization curve — the increasing window radius doesn't monotonically provide a better result. The recall error and difference are minimal at window radius equals 4, while the precision error is relatively flat. By analyzing our situation: in the first half of window radius's growth, more details can be found by slightly increasing the searching range, meanwhile some other non-crane features are included accidentally. In the second half of window radius's growth, it becomes messed up with too big searching range. The patch match is executed too carefully and cautiously. So, less details are confirmed as tower crane but also with less mistakes.

When window radius increases to 6, which matches the second one from left of Figure 4.12, the photometric depth-map becomes more complete, and its corresponding geometric depth-map is over-filtered — more details on the fine structure are treated as outliers. With further growth on window radius, which matches the right half part of Figure 4.12, everything goes uncontrollably bad — pixels on both geometric and photometric depth-maps become intermittent and gather at some spots heavily together. This phenomenon just verifies our inference by analyzing the error in number.

There is a more intuitive comparison between the point-clouds with low (win_r=2) and high (win_r=14) window radius in Figure 4.13. We choose the smallest and the biggest window radius to show easily perceived difference. There is more information than just numbers about how this parameter influences the point-cloud. A significant improvement on surfaces takes place, which makes them more complete. The point-cloud by higher window radius doesn't seem to be bad at all in static, but if we rotate this model or zoom in as in Figure 4.14, we can see some nonsensical strange parts especially at nodes of the crane. More details try to become continuous as surface. On one side, the left image delivers a cleaner and more distinct lattice boom by low window radius. But on the other side, the cabin of the tower crane in the right image looks much better.

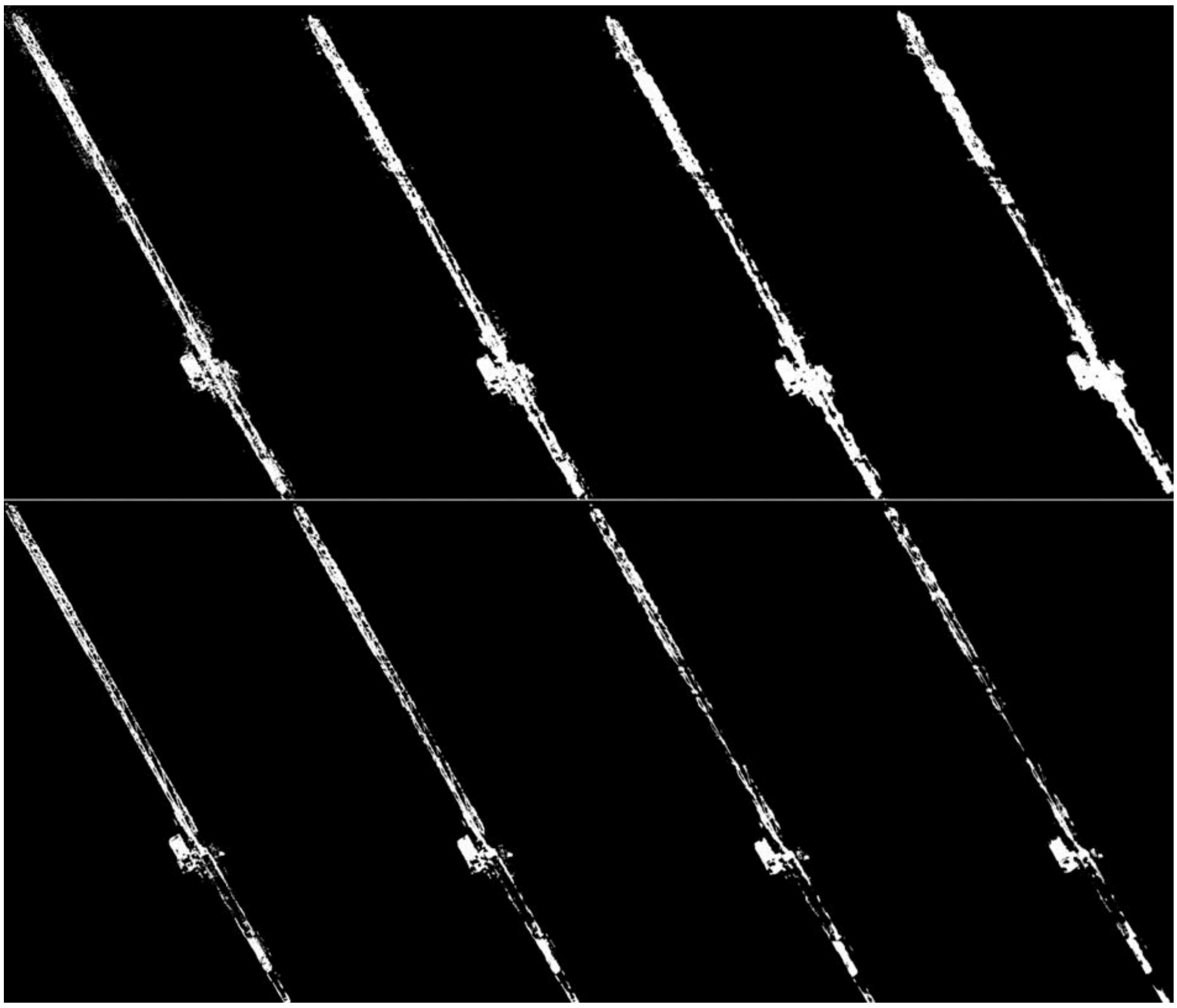


Figure 4.12: Photometric and geometric depth-maps for different window radius. From left to right: 3, 6, 10, 14.

It's hard to define which point-cloud is better. Although the result with window radius equals 3 delivers a much better depth-map than it equals 14, there are some extra details in the latter. Our evaluation script judge the former as the better one, as it has much lower difference compared with the ground-truth and covers more details on the lattice. But in real application, the latter may provide more visible features to human eyes.

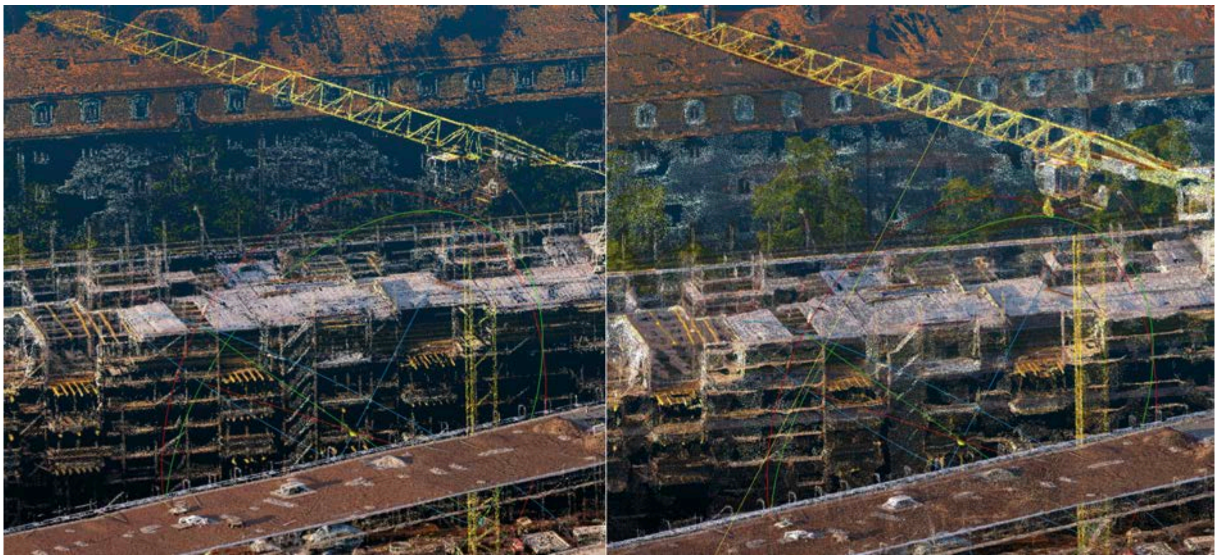


Figure 4.13: Point-clouds with window radius of 2 (left) and 14 (right).

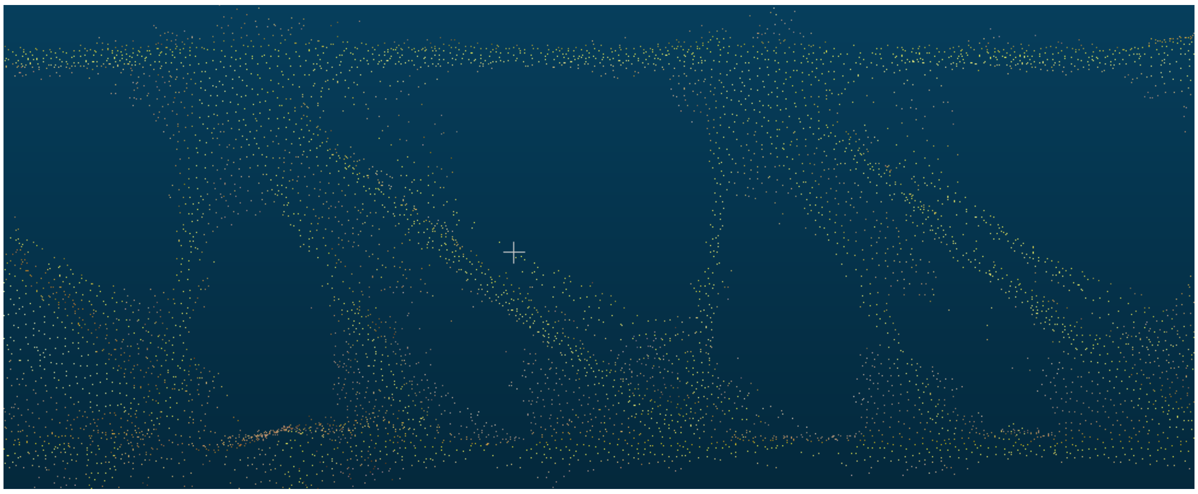


Figure 4.14: Point-cloud with window radius of 14 (zoomed in).

Another thing in Table 4-11 cannot be neglected, the elapsed time arises significantly when the window radius increases. It took more than half day for 39 images on the server, not to mention the time for our whole dataset. A larger window radius can be combined with a bigger window step to reduce the computation time. As for window step, it is the number of pixels to skip when computing NCC. For a value of 1, every pixel is used to compute the NCC. [41] Not all combinations of window sizes and steps produce nice results, especially if the step is greater than 2. Here we only test the case with $window\_step$ = 2 to verify its trade-off of efficiency for quality.

The result in Table 4-12 shows a reasonably decline in both precision and recall, which verifies the negative influence on accuracy by increasing the window step. As for elapsed time in patch match, the window step is supposed to improve efficiency for modeling with a big window radius. Unexpectedly, it didn't save on time in our case.

But it can also be possible that the computation with doubled window step was influenced by some other processes at that period.

Table 4-12: Pixelwise comparison with different window step. (window_radius = 3)

| Win_step | Precision error | Recall error | Difference | Time |
|---|---|---|---|---|
| 1 | 13271 | 51422 | 64693 | 34.3 |
| 2 | 13251 | 52253 | 65504 | 46.4 |

## 4.5 Results on Complete Dataset

As the patch match takes long time, we do less experiments with the whole data at last. Through the test, we can see how the discussed parameters influences on a bigger dense model. And we can also compare the eventual depth-map and point-cloud by same settings with the one generated by smaller dataset before.

The numerical results are listed in Table 4-13. We can see how long the larger dataset takes in comparison with the smaller one with parameters unchanged. In the analyzation before, we only focus on geometric depth-maps because photometric depth-maps have too much noise which may cause strong oscillation. Here, we evaluate the photometric depth-maps from both datasets to check if they are exactly the same. It's interesting that by increasing more source images doesn't change the photometric depth-maps at all, but the geometric version depth-maps have some slight differences. However, if we look at the point-clouds, they are greatly changed as the increased source images provides more information. In Figure 4-16, few details and some outliers are added from left to right on both pairs with different window radius. The upright part of tower crane looks more complete in the larger model, might due to some additional images shot from on the ground. But the price is too expensive for having many white pixels like ghosting around structures. This phenomenon is more apparent in Figure 4.17 under magnification — the contour of lattice is much cleaner by previous selected dataset, but more details are provided from the side view.

The negative effect brought by a large number of images has to be considered. As shown in Figure 4.15, some strange points appear radially around the actual model. They are sparser than the points in construction site but take up a lot of space in the 3D coordinate system. The reason behind could be similar with the reason for white ghosting around the tower crane. One solution is to remove some extra images, which bring very unclear information. Like happened in Chapter 4.4.2, as the number of images increases, it doesn't monotonically improve the quality. Another solution is to set stricter requirements to filter out more features. As the depth-map after patch match stage doesn't change much by increasing more images, there should be some optimization with regard to stereo fusion. Besides, by trying with less automatically involved overlapping images, e.g. setting `__auto__, 5`, doesn't work ungracefully for this big dataset due to the lack of enough correspondences.

Table 4-13: The numerical values on depth-map and evaluation by pixelwise comparison. To spare space in table, 'S' stands for small dataset (39 images) and 'L' for full dataset (472 images).

| Data | Time | Win_r | Type | Mean | Max | Min | Precision | Recall | Difference |
|---|---|---|---|---|---|---|---|---|---|
| S | 26 | 2 | geo | 5.56 | 76.58 | 0.0 | 9875 | 59380 | 69255 |
| | | | pho | 8.14 | 370.22 | -456.08 | 35171 | 39407 | 74578 |
| L | 975 | 2 | geo | 5.56 | 76.73 | 0.0 | 9873 | 59709 | 69582 |
| | | | pho | 8.14 | 370.22 | -456.8 | 35171 | 39407 | 74578 |
| S | 302 | 8 | geo | 7.23 | 702.86 | 0.0 | 14315 | 58242 | 72557 |
| | | | pho | 7.92 | 709.17 | -36.27 | 41591 | 21174 | 62765 |
| L | 1952 | 8 | geo | 7.23 | 689.03 | 0.0 | 14220 | 58037 | 72257 |
| | | | pho | 7.92 | 709.17 | -36.27 | 41591 | 21174 | 62765 |

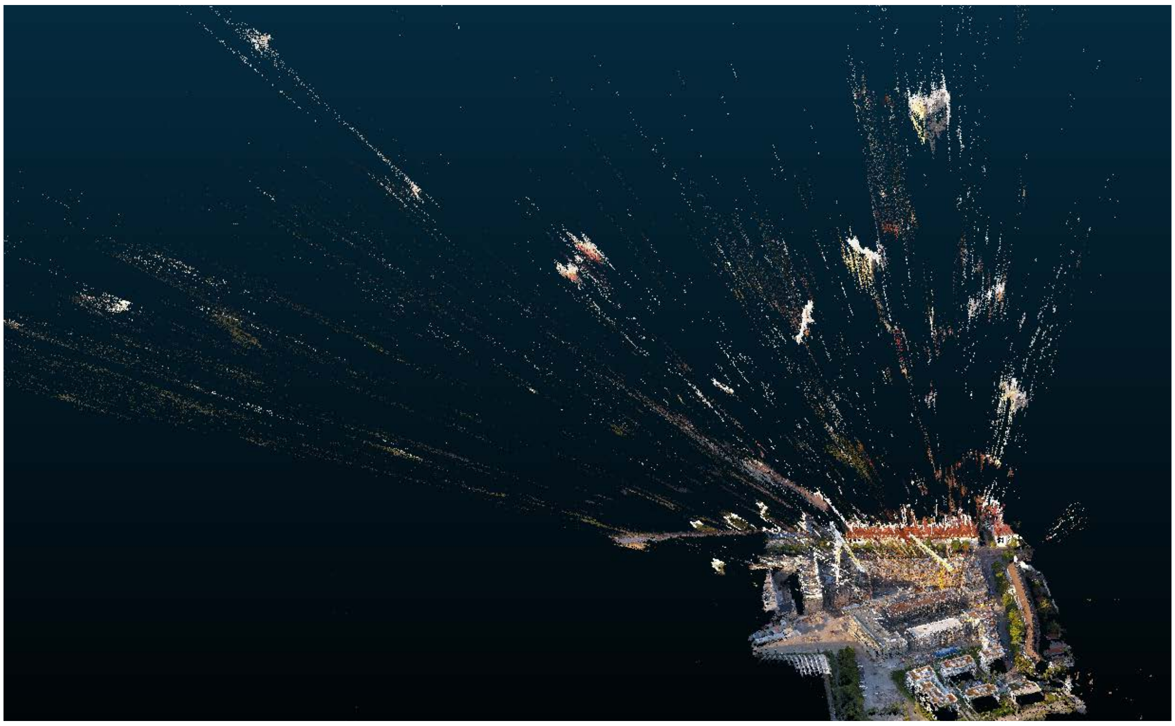


Figure 4.15: Radial outliers on point-cloud of full dataset, window_radius = 8.

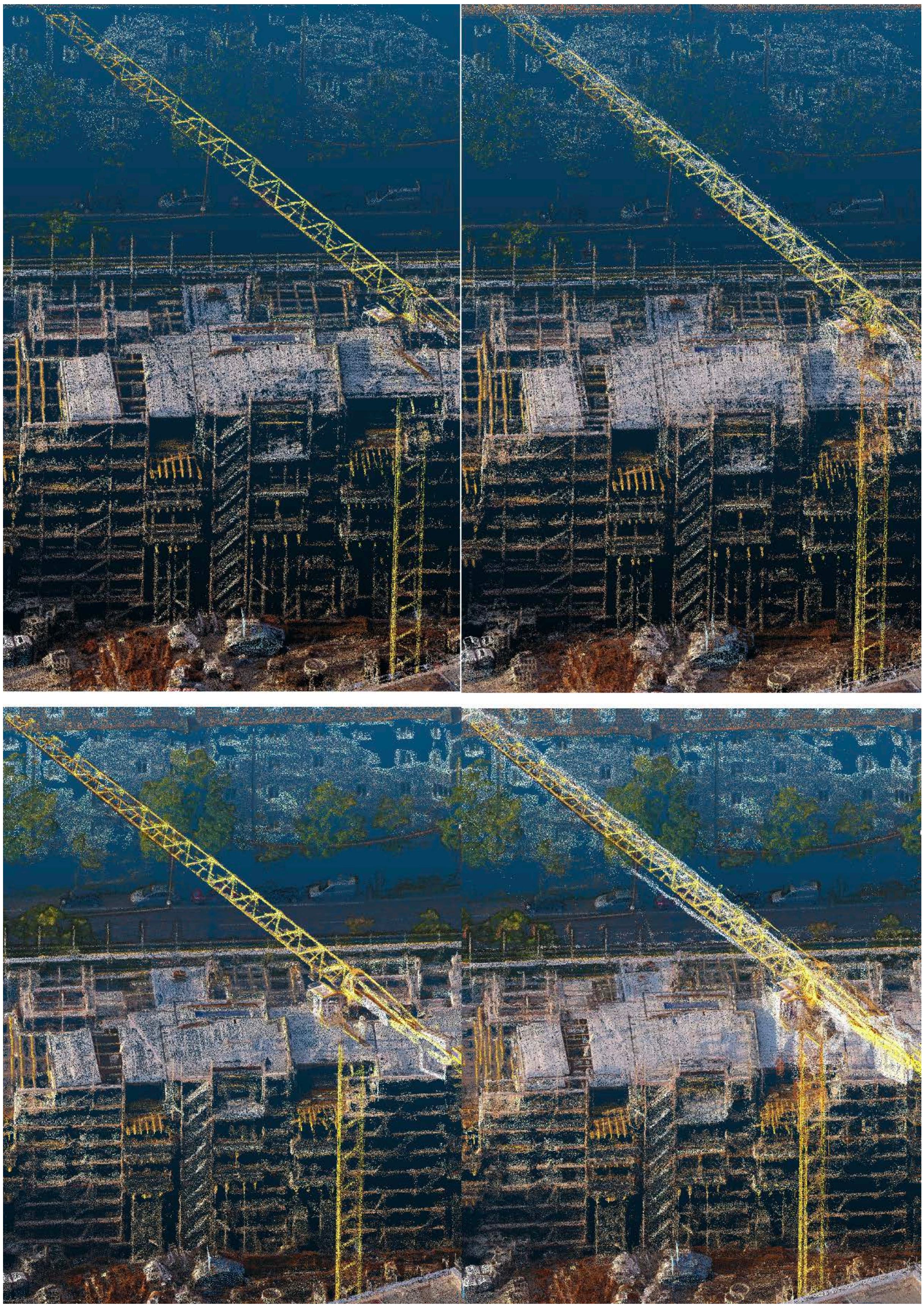


Figure 4.16: Point-clouds with window radius of 2 (up) and 8 (down).

Comparison between models from selected dataset (left) and full dataset (right).

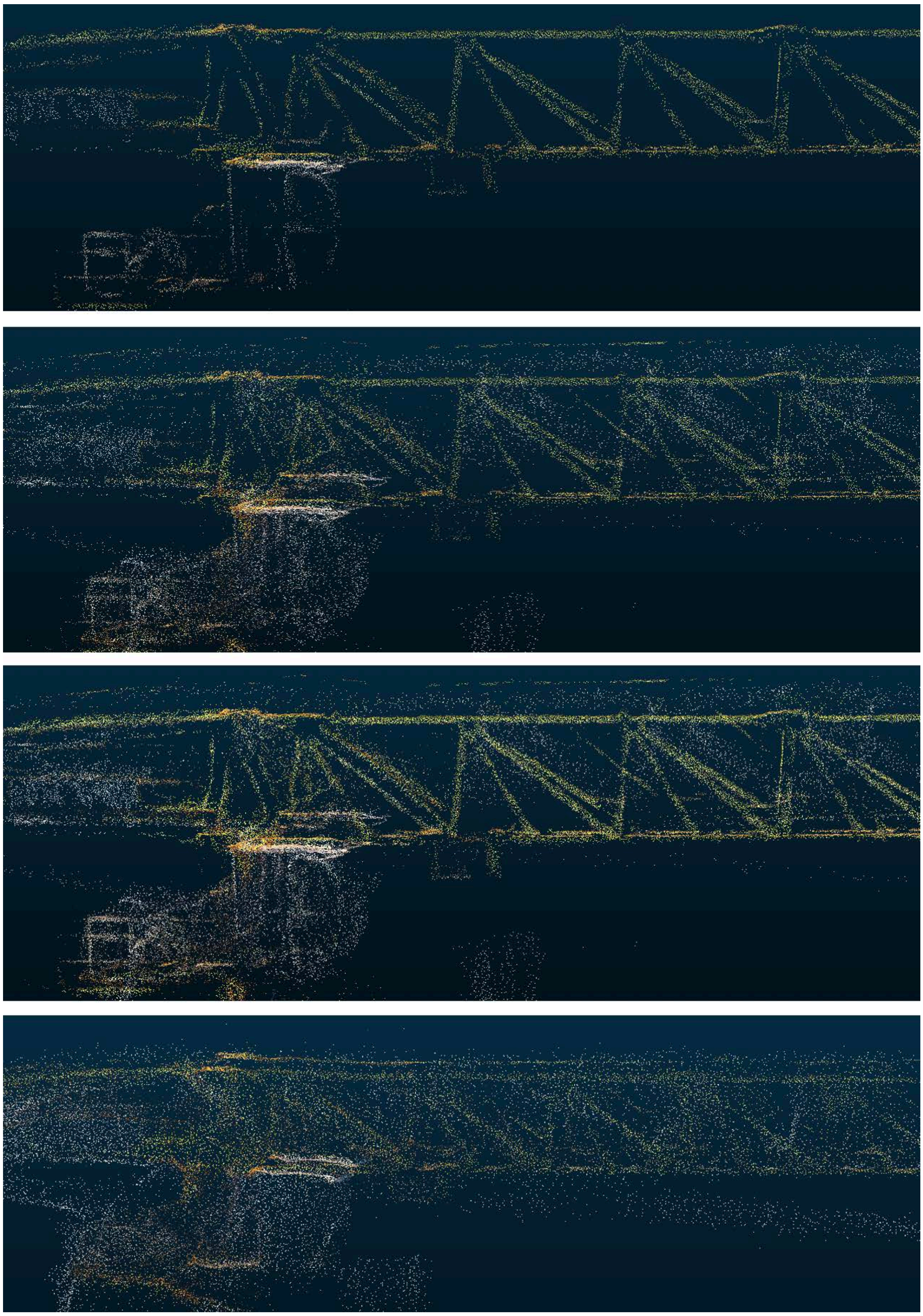


Figure 4.17: Point-clouds with window radius of 2. From top to bottom: geometric (small dataset), photometric (full dataset), geometric (full dataset), default automatic reconstruction (full dataset).

# 5 Discussion and Outlook

## 5.1 Non-Parameter Influences

Except diverse parameters in patch match stage and the whole workflow of Colmap, there are some other factors difficult to quantify but play an important role in the reconstruction. The quality of reconstruction can be guaranteed or even improved if we pay attention to these factors.

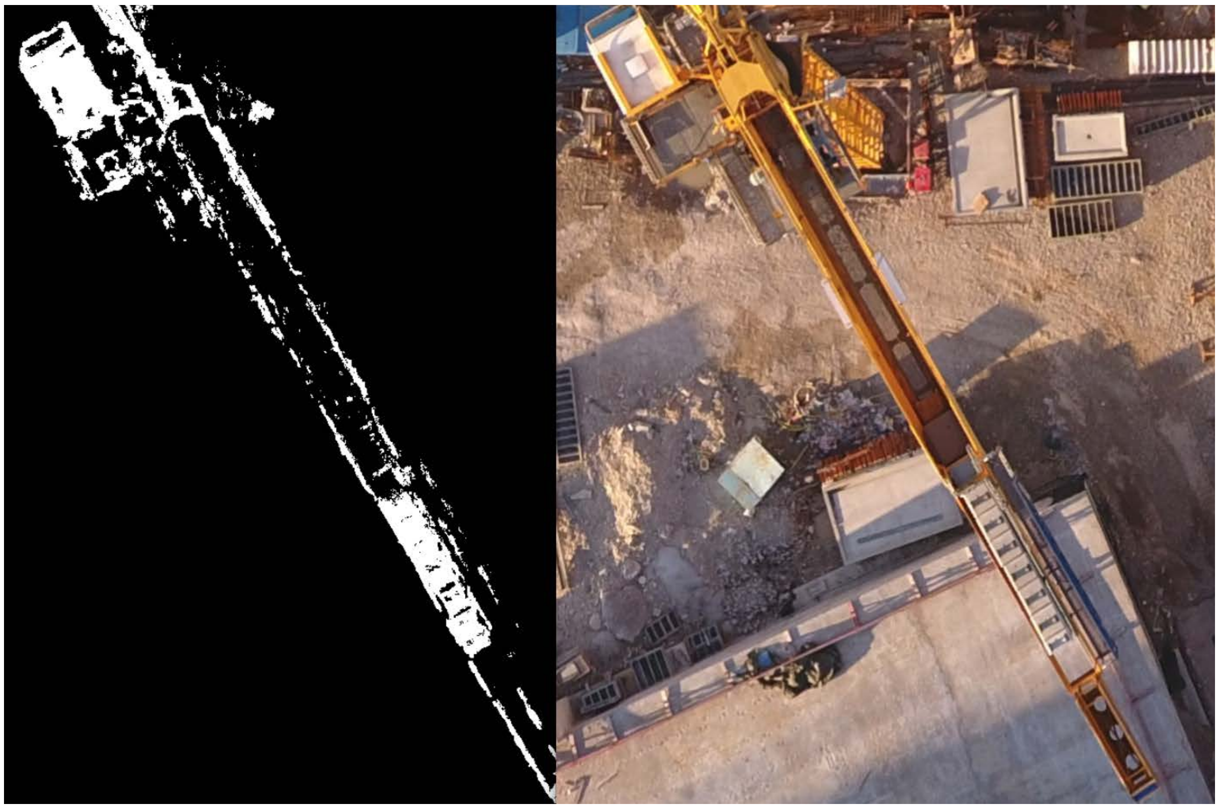


Figure 5.1: Bad reconstruction in the shadows (geometric depth-map).

The chosen image DJI_0038.JPG is very representative with regard to its comprehensive appearance under different influences which can stand for real situation. The sun illuminates a part of the tower crane and buildings and leaves irregular shapes of shadow. These areas with high or low brightness increase the risk of wrong detection. Like discussed in [23], complex shadows would be harmful for detection. And in our study, it also shows a bad recovery for pixels in shadow, as shown in Figure 5.1. The shadows are rather unrecognizable and Colmap usually chooses to go the safe way by rejection. Both tower crane and scaffoldings have strong and complex shadows especially under a sunny weather. To avoid this, the photos should better be captured when cloudy and during a relatively short period, so that the shadows changes less and are more regular for the algorithm to manage.

Additionally, trees also hinder the reconstruction work heavily, especially under the sunshine. In Figure 5.2, we can see that either the details of crane come together with many spots of tree in photometric depth-map or they are removed with these noises in geometric. A different treatment of these features is hardly possible since they are almost in a same scale. This problem can be lightened by changing some camera view point to avoid overlapping trees and desired object in images.

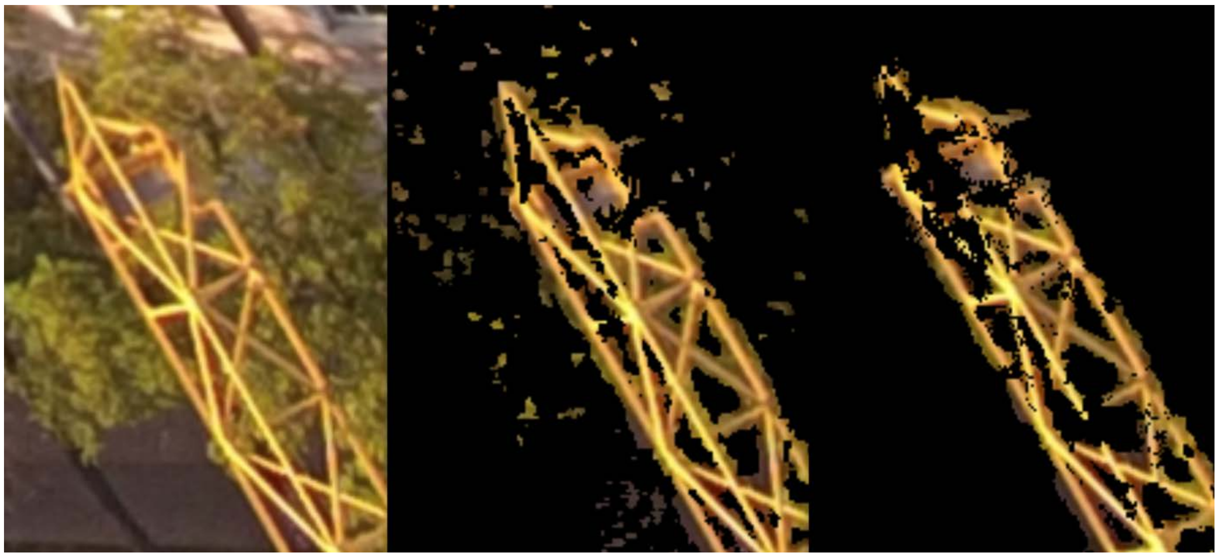


Figure 5.2: From left to right: undistorted image, with photometric mask, with geometric mask.

As for the image quality, it seems at first sight that the more resolution an image has, the more details it provides. But this is not always possible due to the exponentially-growing computation workload and even hardware restriction for excessive resolution.

Under same resolution, sharper the images are, a better reconstruction we can then expect. However, there is a very important phenomenon to mention for this experiment — purple fringing. This effect is usually attributed to chromatic aberration (CA). It is not a new term for people who are familiar with photography. We meet this phenomenon very often when we shoot under strong sun light with backlight or sidelight. This optical aberration is generally most visible as a coloring and lightening of dark edges adjacent to bright areas of broad-spectrum illumination, such as daylight. [43] All these conditions for existence of purple fringing are fulfilled in our case — ample sunlight, fine elements on the crane and a compact camera on the drone.

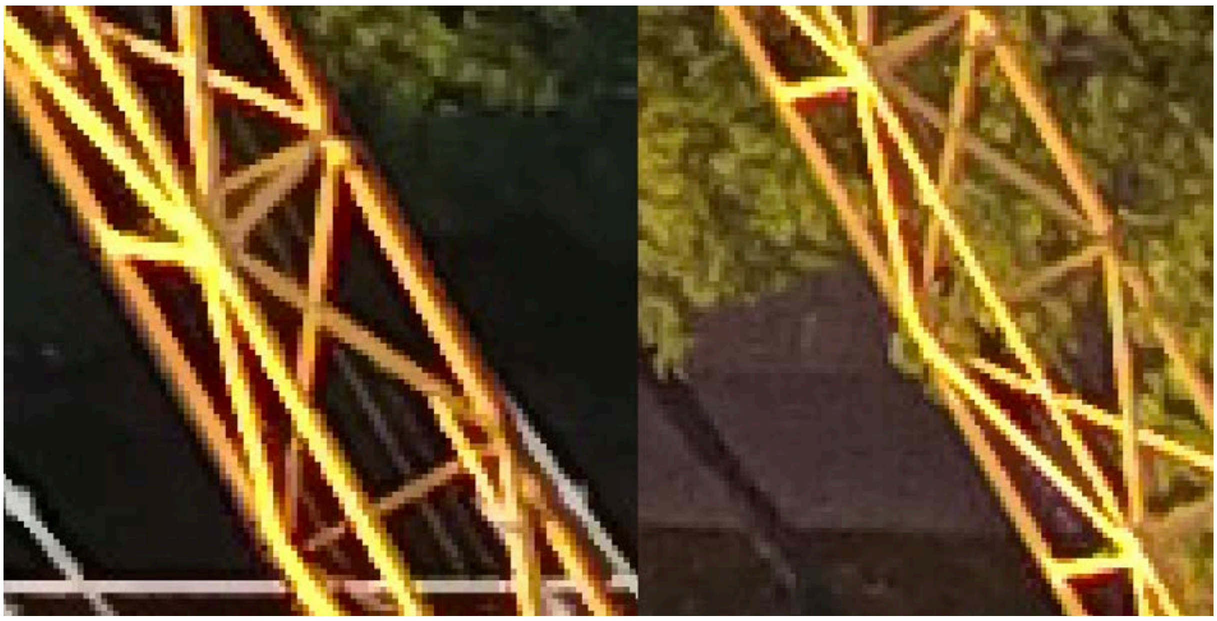


Figure 5.3: Visible purple fringes on the edge of the crane elements.

As shown in the Figure 5.3, some crane elements suffer purple fringing badly. This phenomenon produces the additional colorful pixels on and near the crane elements, which mislead the Colmap into detecting extra features. Even by manual labelling, it becomes harder to recognize a clear boundary. The elements with purple fringing would become either thinner or thicker. Moreover, these features vary on each image randomly, as the camera position changes. As a result, Colmap cannot trust these non-discriminating features and have to drop them or shift them to wrong depth.

A general defocus of the shortest wavelengths resulting in a purple fringe on all sides of a bright object is the result of an axial or longitudinal chromatic aberration. [43] As the purple fringing depends on f-number, what we can do to reduce axial aberration is to use a larger f-number (smaller aperture). Some high-quality heavy camera lenses have some specific elements to reduce this effect. As for drone, this problem will gradually be lightened with the advancement of the camera. And for a better reconstruction, the drone could shoot under cloudy weather and closer for these fine structures to deliver a better result.

## 5.2 Some Remarkable Unwilling Errors during Operation

Some of the errors comes very surprisingly and can hinder the work implicitly.

**1.** The remaining output in depth-map will not be overwritten if Colmap execute the patch match again. The safe way to prevent a wrong analyzation on unwilling former result should be avoided by manually clean-up in time.

**2.** On the local laptop, although the actual image size is 4000 x 2250, the parameter for max image size can't be set to 8000. Otherwise Colmap will throw an error that an illegal memory access was encountered.

**3.** Normally, several containers could run simultaneously. By querying with nvidia-smi, we can know each patch match process use around 4 - 5 GB of GPU. But the server is not always available in full power, since other processes might occasionally occupy the GPU computation. As a result, some elapsed times from the log of Colmap can oscillate heavily at times.

## 5.3 Restrictions and Next Step

The proposed stencil shrinks the area to for the tower crane, which also restrict the evaluation not based on the whole quality of dense model but only the quality of crane. Although the tower crane can be a representative object to reflect the quality of reconstruction, an optimization with regard to the only crane may harm some other surroundings. To overcome this problem, we can have multiple labelled images with a tower crane or something else. An optimization may not lead to improvement on every object, and should thus be voted by most labelled images having positive results.

A more important restriction of evaluation is due to the bitwise comparison itself. Comparing the number of pixels as error is rough and straight-forward. Sometimes the result in number shows a decline, but the point-cloud is not as bad as estimated.

Even the ground-truth is not perfect, and its contour is always a little ambiguous. If we have a nearly perfect software for reconstruction, which means the automated mask is very similar with the ground-truth, counting the number of different pixels would then lead to heavier oscillation.

Some pixels on the edge of fine structure are actually dispensable, but the pixels on the central axis cannot be missed. A more convincing comparison could be a different weight system for pixels in depth-map. Imagine we have two masks of "ground-truth" like in Figure 5.4, where the first ground-truth holds the most important pixels that are definitely a part of the structure, and the second ground-truth has a more tolerant range. The difference of these two masks of ground-truth are treated as gray area. If two masks only differs in gray area, the error will be multiplied with a reduction factor. Any pixels that are missed in the first ground-truth or superfluous out of the second ground-truth have more significant impact on the decline of quality and are thus marked as true error. In the situation for tower crane, it basically means the detected shape can be a little thinner or thicker but can't be intermittent. And the influence by purple fringing would also be minimized. While the price for this concept is a double workload for image-labelling.

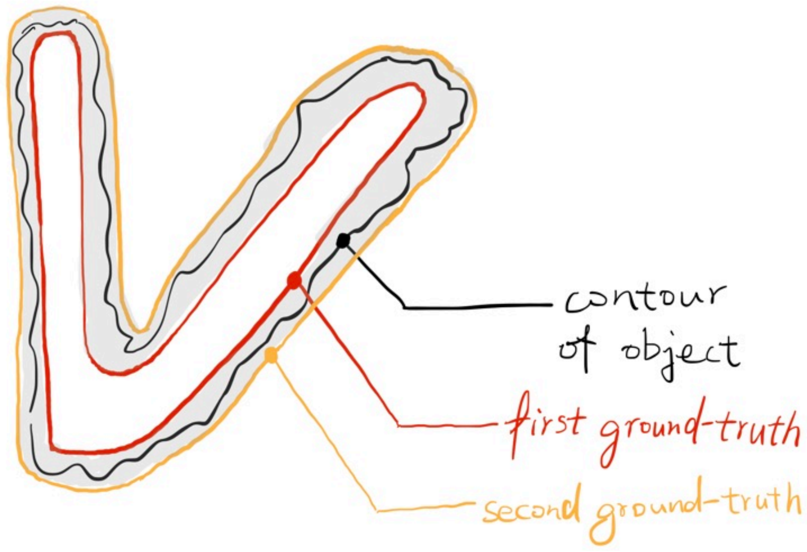


Figure 5.4: An illustrative sketch for two ground-truth evaluation system

There are still much more than discussed in this thesis. We only touched the patch match stage, and even in this stage there are many other parameters to study, not to mention the other stages of Colmap. To overcome the incompatible resolution and unpredictable range of depth due to the modification of SfM, an intelligent algorithm is needed for extracting the tower crane out of the depth-map. The image with slightly different resolution should be aligned or re-distorted to match with ground-truth. And due to the continuous straight-line shape of crane, Hough Transformation [44] might be appropriate for detecting some line segments.

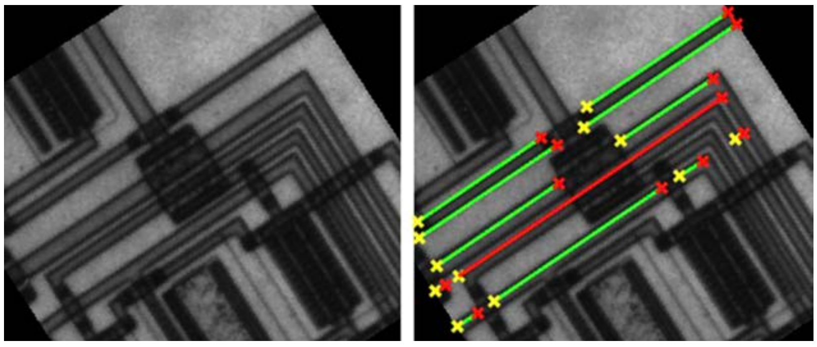


Figure 5.5: Use Hough Transform to detect lines.

## References

[1] Furukawa Y. (2014) Photo-Consistency. In: Ikeuchi K. (eds) Computer Vision. Springer, Boston, MA. https://doi.org/10.1007/978-0-387-31439-6_204

[2] Han, Xian-Feng & Laga, Hamid & Bennamoun, Mohammed. (2019). Image-based 3D Object Reconstruction: State-of-the-Art and Trends in the Deep Learning Era.

[3] R. Hartley and A. Zisserman, Multiple view geometry in computer vision. Cambridge university press, 2003.

[4] Fassi, Francesco & Fregonese, Luigi & Ackermann, Sebastiano & Troia, V. (2013). Comparison between laser scanning and automated 3d modelling techniques to reconstruct complex and extensive cultural heritage areas. ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences. XL-5/W1. 10.5194/isprsarchives-XL-5-W1-73-2013.

[5] Schönberger, Johannes & Frahm, Jan-Michael. (2016). Structure-from-Motion Revisited. 10.1109/CVPR.2016.445.

[6] n16330 Requirements for Point Cloud Compression ISO JCT1 SC29 WG11 Geneva June 2016

[7] Lowe, D.G. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision 60, 91–110 (2004). 10.1023/B: VISI.0000029664. 99615.94

[8] Lindeberg, T. (1993). Scale-Space Theory in Computer Vision

[9] http://www.cse.psu.edu/~rtc12/CSE486/lecture10.pdf

[10] https://medium.com/@vad710/cv-for-busy-devs-improving-features-df20c3aa58-87

[11] https://aishack.in/tutorials/sift-scale-invariant-feature-transform-keypoint-orientation/

[12] J. Dong and S. Soatto, "Domain-size pooling in local descriptors: DSP-SIFT," 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, 2015, pp. 5097-5106, doi: 10.1109/CVPR.2015.7299145.

[13] Y. L. Boureau, J. Ponce, and Y. LeCun. A theoretical analysis of feature pooling in visual recognition. In Proc. of the International Conference on Machine Learning (ICML), pages 111–118, 2010.

[14] Y. Jia, C. Huang, and T. Darrell. Beyond spatial pyramids: Receptive field learning for pooled image features. In Proc. of the Conference on Computer Vision and Pattern Recognition (CVPR), pages 3370–3377, IEEE, 2012.

[15] C. Beder and R. Steffen. Determining an initial image pair for fixing the scale of a 3d reconstruction from an image sequence. Pattern Recognition, 2006.

[16] B. Triggs, P. F. McLauchlan, R. I. Hartley, and A. Fitzgibbon. Bundle adjustment a modern synthesis. 2000.

[17] Schönberger, Johannes & Zheng, Enliang & Pollefeys, Marc & Frahm, Jan-Michael. (2016). Pixelwise View Selection for Unstructured Multi-View Stereo. 9907. 10.1007/978-3-319-46487-9_31.

[18] https://colmap.github.io/tutorial.html#multi-view-stereo

[19] Zheng, E., Dunn, E., Jojic, V., Frahm, J.M.: Patchmatch based joint view selection and depthmap estimation. In: CVPR. (2014)

[20] https://en.wikipedia.org/wiki/Lambertian_reflectance

[21] Ikeuchi, Katsushi (2014). "Lambertian Reflectance". Encyclopedia of Computer Vision. Springer. pp. 441–443. doi:10.1007/978-0-387-31439-6_534. ISBN 978-0-387-30771-8.

[22] (2017). From Differential Photometric Consistency to Surface Differential Geometry Anonymous CVPR submission.

[23] Valle, Eduardo & Picard, David & Cord, Matthieu. (2009). Geometric Consistency Checking for Local-Descriptor Based Document Retrieval. DocEng'09 - Proceedings of the 2009 ACM Symposium on Document Engineering. 135-138. 10.1145/1600193.1600224.

[24] Galliani, S., Lasinger, K., Schindler, K.: Massively parallel multiview stereopsis by surface normal diffusion. In: ICCV. (2015)

[25] Bleyer, M., Rhemann, C., Rother, C.: Patchmatch stereo-stereo matching with slanted support windows. In: BMVC. (2011)

[26] Valle, Eduardo & Picard, David & Cord, Matthieu. (2009). Geometric Consistency Checking for Local-Descriptor Based Document Retrieval. DocEng'09 - Proceedings of the 2009 ACM Symposium on Document Engineering. 135-138. 10.1145/1600193.1600224.

[27] Yasutaka Furukawa and Carlos Hernández (2015), "Multi-View Stereo: A Tutorial", Foundations and Trends® in Computer Graphics and Vision: Vol. 9: No. 1-2, pp 1-148. http://dx.doi.org/10.1561/0600000052

[28] https://docs.docker.com/get-started/

[29] https://docs.docker.com/storage/

[30] https://github.com/NVIDIA/nvidia-docker

[31] https://colmap.github.io/

[32] https://colmap.github.io/tutorial.html#multi-view-stereo

[33] https://colmap.github.io/tutorial.html#feature-detection-and-extraction

[34] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html

[35] https://stackoverflow.com/a/62985765/10173282

[36] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html

[37] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_core/py_image_arithmetics/py_image_arithmetics.html

[38] https://en.wikipedia.org/wiki/Local_search_(optimization)

[39] MLA. Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.

[40] https://colmap.github.io/faq.html#improving-dense-reconstruction-results-for-weakly-textured-surfaces

[41] https://github.com/colmap/colmap/blob/dev/src/mvs/patch_match.h

[42] https://colmap.github.io/faq.html#manual-specification-of-source-images-during-dense-reconstruction

[43] https://en.wikipedia.org/wiki/Purple_fringing

[44] https://www.mathworks.com/help/images/hough-transform.html

# Appendix A

## Colmap Configuration Parameters[1]

### 1_extraction

```
database_path=$database_path
image_path=$image_path
[ImageReader]
single_camera=false
single_camera_per_folder=true
existing_camera_id=-1
default_focal_length_factor=1.2
mask_path=
camera_model=RADIAL
camera_params=
camera_mask_path=
[SiftExtraction]
use_gpu=true
estimate_affine_shape=true
upright=false
domain_size_pooling=true
num_threads=64
max_image_size=8000
max_num_features=24576
first_octave=-1
num_octaves=6
octave_resolution=5
max_num_orientations=2
dsp_num_scales=10
peak_threshold=0.0066666666666666671
edge_threshold=10
dsp_min_scale=0.16666666666666666
dsp_max_scale=4
gpu_index=-1
```

[1] Here are two configuration files as sample. These parameters values are used for reconstruction in this thesis.

**7_patch_match**

```
workspace_path=$dense_model_path
workspace_format=COLMAP
pmvs_option_name=option-all
[PatchMatchStereo]
geom_consistency=true
filter=true
write_consistency_graph=false
max_image_size=-1
window_radius=$win_r
window_step=1
num_samples=15
num_iterations=$it
filter_min_num_consistent=2
depth_min=-1
depth_max=-1
sigma_spatial=-1
sigma_color=0.20000000298023224
ncc_sigma=0.60000002384185791
min_triangulation_angle=1
incident_angle_sigma=0.89999997615814209
geom_consistency_regularizer=0.30000001192092896
geom_consistency_max_cost=3
filter_min_ncc=$ncc
filter_min_triangulation_angle=3
filter_geom_consistency_max_cost=1
cache_size=200
gpu_index=-1
```

# Appendix B

## Source Code

### B.1 Dockerfile[2]

```
FROM nvidia/cuda:10.2-cudnn7-devel
LABEL maintainer="ripfreeworld@icloud.com"
# "Under newer Ubuntu versions it might be necessary to explicitly select
# the used GCC version due to compatibility issues with CUDA"

# to avoid stalling at selecting the geographic area
ENV DEBIAN_FRONTEND=noninteractive \
    APPROOT="/app"

RUN apt-get update && apt-get install -y \
    git \
    cmake \
    build-essential \
    libboost-program-options-dev \
    libboost-filesystem-dev \
    libboost-graph-dev \
    libboost-regex-dev \
    libboost-system-dev \
    libboost-test-dev \
    libeigen3-dev \
    libsuitesparse-dev \
    libfreeimage-dev \
    libgoogle-glog-dev \
    libgflags-dev \
    libglew-dev \
    qtbase5-dev \
    libqt5opengl5-dev \
    libcgal-dev \
    libcgal-qt5-dev \
```

[2] An official Dockerfile was not released at the time Chenyang used the docker image for Colmap. This Docker image is also public on DockerHub repository: https://hub.docker.com/repository/docker/ripfree-world/colmap_cuda10.2

```
    libatlas-base-dev \
    libsuitesparse-dev \
    software-properties-common \
    xvfb \
    python3 \
    python3-pip \
    libpq-dev \
    python3-dev

WORKDIR ${APPROOT}

RUN git clone https://ceres-solver.googlesource.com/ceres-solver && \
    git clone https://github.com/colmap/colmap.git


WORKDIR ${APPROOT}/ceres-solver
#RUN git checkout $(git describe --tags)
# Checkout the latest release
RUN mkdir build
WORKDIR ${APPROOT}/ceres-solver/build
RUN cmake .. -DBUILD_TESTING=OFF -DBUILD_EXAMPLES=OFF && make -j && make install

WORKDIR ${APPROOT}/colmap/build
RUN cmake ..
RUN make -j
RUN make install

WORKDIR ${APPROOT}
# clean up
RUN rm -r colmap ceres-solver

# for possible use case of PostgreSQL
RUN pip3 install psycopg2

WORKDIR /

```

## B.2 read_bin.py

```python
import numpy as np
import matplotlib.pyplot as plt
import os
import matplotlib
import glob
import time

# to have a better visible result of depth-map, depth-map should be smooth, \
# very big and small values need to be filtered
def filter_crane(nd_array):
    # the mean_array represents a relatively normal distance, from which \
    # the values in the depth-map shouldn't deviate too much
    mean_array = np.mean(nd_array)
    # substitute very big and small values with the mean value, these values are \
    # usually wrong and thus lead to very bad image of depth-map
    # drawback: this loop is very inefficient
    for i, row in enumerate(nd_array):
        for j, column in enumerate(row):
            # not delete outliers, but keep these pixels with values closer to mean
            # if nd_array[i][j] > 5 * mean_array:
            #     nd_array[i][j] = mean_array
            # if nd_array[i][j] < mean_array / 5:
            #     nd_array[i][j] = mean_array

            # e.g. range [6.19, 4.66] is for crane to filter
            if nd_array[i][j] > 6.19:
                nd_array[i][j] = 0
            if nd_array[i][j] < 4.66:
                nd_array[i][j] = 0
    return nd_array

# the function read_array from Author: Johannes L. Schoenberger (jsch@demuc.de)
def read_array(path):
    with open(path, "rb") as fid:
        width, height, channels = np.genfromtxt(fid, delimiter="&", max_rows=1,
                                                usecols=(0, 1, 2), dtype=int)
        fid.seek(0)
        num_delimiter = 0
        byte = fid.read(1)
        while True:
            if byte == b"&":
                num_delimiter += 1
                if num_delimiter >= 3:
                    break
            byte = fid.read(1)
        array = np.fromfile(fid, np.float32)
    array = array.reshape((width, height, channels), order="F")
    return np.transpose(array, (1, 0, 2)).squeeze()

if __name__ == '__main__':
    # input the .bin files under depth_map_bin, copied from the output of Colmap
    # file_names contains multiple files
    # * means get both photometric and geometric
    file_names = glob.glob("colmap_output/DJI_0038.JPG.*.bin")
    for index, file_name in enumerate(file_names):
        time_start = time.time()
        # to have a sub-folder which stores the depth_maps in png format
        if not os.path.exists("depth_map_png"):
            os.makedirs("depth_map_png")
        # take notice of the BGR instead of RGB color order in the depth_map
        # im_bgr = cv2.imread('data/src/lena.jpg')
        # im_rgb = im_bgr[:, :, [2, 1, 0]]
        depth_map = read_array(file_name)
        print(np.mean(depth_map), np.amax(depth_map), np.amin(depth_map))
        filtered_depth_map = filter_crane(depth_map)
        time_end = time.time()
        # show a time cost for each run, especially depth filter
        print('time cost', time_end - time_start, 's')
        if file_name == 'colmap_output/DJI_0038.JPG.photometric.bin':
            pho_or_geo = 'photometric'
        else:
            pho_or_geo = 'geometric'
        # save the .png file with the type information or index_num into the folder
        matplotlib.image.imsave('depth_map_png/DJI_0038_{}.png'.format(pho_or_geo),
filtered_depth_map)
        # print resolution
        print(filtered_depth_map.shape)
        # interactive way to find out depth value on the crane
        plt.imshow(filtered_depth_map)
        plt.show()
```

## B.3 mask_on_image.py

```python
# this mask_on_img.py firstly creates a stencil based on the ground truth,
# and then uses the stencil to restrict the region of interest on the depth-map.
# the folders depth_map_png and ground_truth are input
# the folder region_of_interest is the intermediate output
# the folder mask_by_color is the output in the end
from pathlib import Path
import os
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from reportlab import xrange

# turn the labelled image into mask
def ground_truth_mask(path):
    img = cv.imread(path, cv.IMREAD_UNCHANGED)
    # for the case that the image has the fourth channel for transparency
    if img.shape[2] == 4:
        a1 = ~img[:, :, 3]
        img = cv.add(cv.merge([a1, a1, a1, a1]), img)
        img = cv.cvtColor(img, cv.COLOR_RGBA2RGB)
    gray = cv.cvtColor(img, cv.COLOR_RGB2GRAY)
    # to ENSURE the hand labelled image only has black and white
    ret, mask_img = cv.threshold(gray, 254, 255, cv.THRESH_BINARY_INV)
    if not os.path.exists("ground_truth_mask"):
        os.makedirs("ground_truth_mask")
    # cv.imwrite('ground_truth_mask/test_2.png', mask_img)
    return mask_img

# apply gaussian blur to the mask to expand the borders with a shade
def enlarge_mask(mask_img, kernel_size):
    # the GaussianBlur kernel size must be odd, 99 is big enough
    blur_img = cv.GaussianBlur(mask_img, (kernel_size, kernel_size), 0)
    # after gaussian blur there are pixels besides 0 and 255 on the border area
    # enlarge: all the pixels with non-zero value should be contained into the mask
(as white area 255)
    ret, new_mask = cv.threshold(blur_img, 0, 255, cv.THRESH_BINARY)
    return new_mask

```

```python
# shrink the depth-map with stencil, output to region_of_interest
def apply_stencil(truth_path, depth_path, index, kernel_size, output_path):
    if not os.path.exists("region_of_interest"):
        os.makedirs("region_of_interest")
    # ground truth is the mask by labelling
    ground_truth = ground_truth_mask(truth_path)
    depth_map = cv.imread(str(depth_path), 0)
    # stencil mask is an enlarged mask based on the ground truth
    stencil_mask = enlarge_mask(ground_truth, kernel_size)
    roi = cv.bitwise_and(depth_map, depth_map, mask=stencil_mask)
    # the non-crane area were saved as 30 instead of 0
    ret, roi = cv.threshold(roi, 40, 255, cv.THRESH_BINARY)

    # to save the image under stencil to local
    cv.imwrite(str(output_path).format(index), roi)
    return depth_map, ground_truth, stencil_mask, roi


# display original img, ground truth, stencil and roi (after stencil)
def display_stencil(images):
    titles = ['original depth map', 'groundtruth mask', 'stencil', 'region of in-
terest']
    for i in xrange(4):
        plt.subplot(2, 2, i + 1), plt.imshow(images[i])
        plt.title(titles[i])
        plt.xticks([]), plt.yticks([])
    plt.show()


# display image and mask in a window
def display_mask(img, mask, l_limit, u_limit):
    extracted_color_img = cv.bitwise_and(img, img, mask=mask)
    titles = ['Original Image ROI', '{} ~ {}'.format(l_limit, u_limit), 'Image af-
ter mask']
    images = [img, mask, extracted_color_img]
    for i in xrange(3):
        plt.subplot(2, 2, i + 1), plt.imshow(images[i])
        plt.title(titles[i])
        plt.xticks([]), plt.yticks([])
    plt.show()


# to show a remaining undistorted image after depth range for crane applied
def mask_undistorted(path_depth_crane_range, path_undistorted_image, output_path):
    depth_crane_range = cv.imread(path_depth_crane_range, 0)
    undistorted_image = cv.imread(path_undistorted_image)
    ret, depth_crane_mask = cv.threshold(depth_crane_range, 40, 255, cv.THRESH_BI-
NARY)
    crane_range = cv.bitwise_and(undistorted_image, undistorted_image,
mask=depth_crane_mask)
    cv.imwrite(output_path, crane_range)


def mask_crane(output_path):
    # mask_list stores the corresponding mask of detection
    mask_list = []
    if not os.path.exists("mask_by_color"):
        os.makedirs("mask_by_color")
    # file access mode: read only
    file_threshold = open(r"color_threshold.txt")
    # read all the lines in the .txt file as a list
    threshold_list = file_threshold.readlines()
    for index, line in enumerate(threshold_list):
        fields = line.split("; ")
        # to set lower and upper color limits, e.g. np.array([125, 0, 0])
        lower_limit_str = fields[0]
        # this fields[0] is not [125, 0, 0] but ['125 0 0'], i.e. string
        # split the numbers into a list
        l_list = lower_limit_str.split()
        # map(function, iterable),
        map_l = map(int, l_list)
        lower_limit = np.array(list(map_l))
        upper_limit_str = fields[1]
        r_list = upper_limit_str.split()
        map_r = map(int, r_list)
        upper_limit = np.array(list(map_r))
        # load image as colorful, note: BGR instead of RGB by default
        # update 2020-11-14: use grayscale to easier select
        img = cv.imread('region_of_interest/roi_nov15_{}.png'.format(index), 0)
        # threshold the BGR to get only a certain color(distance) range
        # update 2020-11-15: the crane is much brighter than background in stencil
        # crane range: about 180, background range: about 30
        the_mask = cv.inRange(img, 40, 255)
        cv.imwrite(output_path.format(index), the_mask)
        mask_list.append(the_mask)
        return mask_list


if __name__ == '__main__':
    file_path = Path('/home/lcy/Desktop/results/last/')
    # it depends on how many pairs of depth_map and ground_truth we have
```

```
  for i in xrange(1):
    # update 2020-11-15: this depth-map is already filtered by range for crane
    geo_depth_map_path = str(file_path / 'depMap_geometric.png')
    pho_depth_map_path = str(file_path / 'depMap_photometric.png')
    ground_truth_path = 'ground_truth/groundtruth_2.png'
    stencil_geo_path = str(file_path / 'stencil_geo_{}.png')
    stencil_pho_path = str(file_path / 'stencil_pho_{}.png')
    # Gaussian Blur kernel size must be odd, e.g. 99
    stencil_pair_geo = apply_stencil(ground_truth_path, geo_depth_map_path, 0,
99, stencil_geo_path)
    stencil_pair_pho = apply_stencil(ground_truth_path, pho_depth_map_path, 0,
99, stencil_pho_path)
    # display_stencil(stencil_pair)
    # undistorted_image_path = 'real_image/DJI_0038.JPG'
    # mask_undistorted_ot_path = 'undistorted_isolat/crane_range_nov15_geo.png'
  #  mask_undistorted(depth_map_path, undistorted_image_path, mask_un-
distorted_ot_path)
  #
  # ot_path = 'mask_by_color/mask_auto_nov15_{}.png'
  # mask_crane(ot_path)
```

### B.4 image_compare.py

```python
import numpy as np
import cv2 as cv
from PIL import ImageFilter
from reportlab import xrange
import matplotlib.pyplot as plt
from pathlib import Path


def diff_subtract_images(image_detected, image_groundtruth):
    # pay attention the non-absolute subtraction
    difference = cv.absdiff(image_groundtruth, image_detected)
    intersection = cv.bitwise_and(image_groundtruth, image_detected)
    union = cv.bitwise_or(image_groundtruth, image_detected)
    # countNonZero: the subtraction happens regardless of the order A - B or B - A,
do bitwise_or to prevent negative
    precision_error_pixels = union - image_groundtruth
    precision_error = cv.countNonZero(precision_error_pixels)
    print(cv.countNonZero(image_groundtruth), cv.countNonZero(precision_error_pix-
els), cv.countNonZero(image_detected))
    # recall is about how much of the ground truth is detected
    # do bitwise_or to prevent negative
    recall_error_pixels = image_groundtruth - intersection
    print(cv.countNonZero(recall_error_pixels))
    recall_error = cv.countNonZero(recall_error_pixels)
    diff_pixels = cv.countNonZero(difference)

    return difference, precision_error_pixels, recall_error_pixels, diff_pixels,
precision_error, recall_error


def read_mask(mask_color_iso_path, mask_hand_label_path):
    mask_color_iso = cv.imread(mask_color_iso_path, 0)
    mask_hand_label = cv.imread(mask_hand_label_path, 0)
    # make sure all values are either 255 or 0
    ret, mask_color_iso = cv.threshold(mask_color_iso, 0, 255, cv.THRESH_BINARY)
    ret, mask_hand_label = cv.threshold(mask_hand_label, 0, 255, cv.THRESH_BINARY)
    return mask_color_iso, mask_hand_label


def display_bitwise_comparison(diff_img, detected, recall):
    fig = plt.figure("Images")
    images = ("difference_of_images", diff_img), ("wrong in detection", detected),\
             ("ground truth not found", recall)
    for (i, (name, image)) in enumerate(images):
        # show the image
        ax = fig.add_subplot(1, 3, i + 1)
        ax.set_title(name)
        plt.imshow(image, cmap=plt.cm.gray)
        plt.axis("off")


# to evaluate the results on representative depth-map
# input: path_mask_by_color, path_ground_truth, or by default
def evaluate():
    # the error_list stores all the difference of each pair of masks
    error_list = []
    file_path = Path('/home/lcy/Desktop/results/last/')
    # evaluate all (10) sample depth maps comparing with the ground truth
    for index in xrange(1):
        # mask_auto is the results from mask generation
        mask_auto, mask_truth = read_mask(str(file_path / 'stencil_pho_{}.png').\
                    format(index), 'ground_truth_mask/test_2.png')
        # MSE_gray_orig = mse(img_gray, img_original)
        diff_image, wrong_detected, wrong_recall, difference, precision_ratio, \
                    recall_ratio = diff_subtract_images(mask_auto, mask_truth)
        # print the proportion of the wrong detected area
        print("The difference is:", difference)
        print("the precision error is:", precision_ratio)
        print("The recall error is:", recall_ratio)
        display_bitwise_comparison(diff_image, wrong_detected, wrong_recall)
    # it returns a list of error, there should be a way to compare several list of
errors
    # 1: compare the mean error with another iteration, how much better
    # 2: compare each error respectively and see how many of them are improved or
declined
    # e.g. when 8 among 10 are improved, and mean error very slightly declined,
consider this as an improvement
    return error_list


if __name__ == '__main__':
    evaluate()

```

## B.5 optimize.py[3]

```python
import random
import app
from statistics import mean
import imageCompare
from pathlib import Path

# to call the Colmap software for certain step(s)
def run_colmap(para_dict):
    project_dir = Path('./')
    config_target = project_dir / "tconfig_new"
    # the parameters not listed are using default values
    reconstruction_configuration = app.ReconstructionConfig.\
             CreateStandardConfig(root_dir=project_dir, pm_num_iteration=\
                                 window_radius=para_dict['window_r'][-1] if\
                                 para_dict['window_r'] else None)
    reconstruction_configuration.ply_output_path = project_dir
    # specify the step to run e.g. tconfig/1_extraction
    job_file_path = config_target / "7_patch_match"
    dir_of_this_file = app.Reconstructor.GetPathOfCurrentFile()
    # load the tconfig folder
    config_source = Path(dir_of_this_file.parent / "tconfig")
    app.Reconstructor.Generic2SpecificJobFiles(config_source, config_target, \
             reconstruction_configuration)
    # execute only the job described by file name 1_extraction -> feature extractor
    app.Reconstructor.execute_job(job_file_path, reconstruction_configuration)

def get_stepsize(best_parameter, parameter_history, default_stepsize):
    # the best_parameter_value is the last value in this list
    best_parameter_value = best_parameter[-1]
    # to get the index of this bpv in the whole parameter_history
    index_of_bpv = parameter_history.index(best_parameter_value)
    # count the distance to the end
    distance_from_tail = len(parameter_history) - index_of_bpv
    # the distance_from_tail acts as the exponent of "2"
    # the next step_size should always be half of the former step_size
    divisor = 2 ** distance_from_tail
```

[3] This optimize.py is to show the concept of optimization.

```python
    step_size = int(default_stepsize / divisor)
    return step_size


# each run in this function optimize() do an optimization on one parameter
def optimize(parameter_name, the_para_step, para_dict, best_para_dict):
    # read from the_para_step
    lower_bound = the_para_step[0]
    upper_bound = the_para_step[1]
    initial_step = the_para_step[2]
    # counter: how many times it fails to improve, in order to pop out
    failures = the_para_step[3]
    # this random value is generated between the both bound values
    new_parameter_value = random.randint(lower_bound, upper_bound)
    # avoid duplicates
    if new_parameter_value not in para_dict[parameter_name]:
        # write this initial_start_value into the para_dict for the first run
        para_dict[parameter_name].append(new_parameter_value)

    while True:
        # run the corresponding step (patch match) of the COLMAP
        run_colmap(para_dict)
        errors = imageCompare.evaluate()
        # errors is a list describing difference of pixels
        list_errors.append(errors)
        # compare if better than the last iteration
        # average error is a naive way to evaluate, better comparison required
        if mean(errors) < mean(list_errors[-1]):
            best_para_dict[parameter_name].append(new_parameter_value)
        else:
            # means it's getting worse, count this as a failure
            failures += 1
            # update the_para_step
            the_para_step[3] = failures
            break

        # HERE SET MAXIMUM OF FAILURES
        if failures > 5:
            para_dict.pop()
            break
        # step_size is positive
        step_size = get_stepsize(best_para_dict[parameter_name], para_dict[parameter_name], initial_step)
        # apply stepsize and get new parameter
        new_parameter_value = para_dict[parameter_name][-1] + step_size
        # if step_size is 0, means the value of this parameter cannot be updated any more in current direction
        if step_size > 0 and new_parameter_value < upper_bound and new_parameter_value not in para_dict[parameter_name]:
            # append the to the parameters history list anyway
            para_dict[parameter_name].append(new_parameter_value)
            # this new value of parameter_dict will be
        else:
            # count this situation as a failure
            failures += 1
            # update the_para_step
            the_para_step[3] = failures
            break


if __name__ == '__main__':
    # the "keys" of dict are parameter_names to optimize
    parameters_dict = {}
    # HERE ARE THE INPUT FOR OPTIMIZATION ON PARAMETERS
    # parameter_step provides min, max, step_size and a counter for failures
    parameter_step = {'window_r': [3, 10, 2, 0]}
    # initialize the dict with empty, indicating the "value" of dict is a list
    parameters_dict['window_r'] = []
    parameters_best_dict = parameters_dict

    # run Colmap with empty parameters_dict, that means using the default values
    run_colmap(parameters_dict)
    # list of list
    list_errors = []
    initial_errors = imageCompare.evaluate()
    list_errors.append(initial_errors)

    # at last, all of the parameters in the list should be popped out
    # pop: remove from parameters_dict but keeps the parameters_best_dict as result
    # if not all popped out, restart
    while parameters_dict:
        # random order, but covers ALL of the parameters
        keys = list(parameters_dict.keys())
        # without explicit list of keys, cannot reshuffle a dictionary
        random.shuffle(keys)
        # each time the list `keys` has a different sequence of parameter names
        for name in keys:
            optimize(name, parameter_step[name], parameters_dict, \
                    parameters_best_dict)

```

## B.6 CFG_generator.py

```python
# In original `.cfg` file we have all the images available for dense
cfg_file = open('patch-match.cfg')
cfg_lines = cfg_file.readlines()
cfg_file.close()

# here we have all the relating images to DJI_0038.JPG
f1 = open('relating.txt')
# only one line in relating.txt
relating_image_names = f1.readline()
f1.close()

# split by 2 spaces, returns list of string
image_names = relating_image_names.split('  ')
# empty string for storing new CFG content
new_cfg = ""

# Each line stand for a certain image, e.g. 'overhead/DJI_0042.JPG\n'
# or the line indicating the number of most visual overlap, \
# e.g. '__auto__, 20\n'
for line in cfg_lines:
    # [-13:-1] because of the last char for '\n'
    image_name = line[-13:-1]

    if image_name in relating_image_names:
        new_cfg += line
         # append most visual overlap automatically as source images
        new_cfg += '__auto__, 20\n'

f2 = open("new_patch_match.cfg", "w")
f2.write(new_cfg)
f2.close()

```

# Declaration of Originality

With this statement I declare, that I have independently completed this Master Thesis. The thoughts taken directly or indirectly from external sources are properly marked as such. This thesis was not previously submitted to another academic institution and has also not yet been published.

Munich, 2. December 2020

First Name Family Name

Name

Address

München

E-mail Address