3D Scene Reconstruction with Micro-Aerial Vehicles and Mobile Devices

Ivan Dryanovski

Graduate Center, City University of New York

9-2015

How does access to this work benefit you? Let us know!

Follow this and additional works at: https://academicworks.cuny.edu/gc_etds

Part of the Robotics Commons

Recommended Citation

https://academicworks.cuny.edu/gc_etds/911

This Dissertation is brought to you by CUNY Academic Works. It has been accepted for inclusion in All Dissertations, Theses, and Capstone Projects by an authorized administrator of CUNY Academic Works. For more information, please contact deposit@gc.cuny.edu.
3D Scene Reconstruction with Micro-Aerial Vehicles and Mobile Devices

by

Ivan Dryanovski

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

2015
Abstract

3D Scene Reconstruction with Micro-Aerial Vehicles and Mobile Devices

by

Ivan Dryanovski

Advisor: Prof. Jizhong Xiao

Scene reconstruction is the process of building an accurate geometric model of one’s environment from sensor data. We explore the problem of real-time, large-scale 3D scene reconstruction in indoor environments using small laser range-finders and low-cost RGB-D (color plus depth) cameras. We focus on computationally-constrained platforms such as micro-aerial vehicles (MAVs) and mobile devices. These platforms present a set of fundamental challenges - estimating the state and trajectory of the device as it moves within its environment and utilizing lightweight, dynamic data structures to hold the representation of the reconstructed scene. The system needs to be computationally and memory-efficient, so that it can run in real time, onboard the platform.

In this work, we present three scene reconstruction systems. The first system uses a laser range-finder and operates onboard a quadrotor MAV. We address the issues of autonomous control, state estimation, path-planning,
and teleoperation. We propose the multi-volume occupancy grid (MVOG) - a novel data structure for building 3D maps from laser data, which provides a compact, probabilistic scene representation.

The second system uses an RGB-D camera to recover the 6-DoF trajectory of the platform by aligning sparse features observed in the current RGB-D image against a model of previously seen features. We discuss our work on camera calibration and the depth measurement model. We apply the system onboard an MAV to produce occupancy-based 3D maps, which we utilize for path-planning.

Finally, we present our contributions to a scene reconstruction system for mobile devices with built-in depth sensing and motion-tracking capabilities. We demonstrate reconstructing and rendering a global mesh on the fly, using only the mobile device’s CPU, in very large (300 square meter) scenes, at a resolutions of 2-3cm. To achieve this, we divide the scene into spatial volumes indexed by a hash map. Each volume contains the truncated signed distance function for that area of space, as well as the mesh segment derived from the distance function. This approach allows us to focus computational and memory resources only in areas of the scene which are currently observed, as well as leverage parallelization techniques for multi-core processing.
“Da steh’ ich nun, ich armer Tor,
Und bin so klug als wie zuvor!”

Johann Wolfgang von Goethe, Faust
Acknowledgements

I would like to thank my advisor, Prof. Jizhong Xiao, for his guidance and support throughout my doctoral studies. He provided me with the tools and flexibility to pursue my research interests, and I could always count on his support and advice in navigating the academic world.

I am grateful to the members of my doctoral committee - Prof. Zhigang Zhu, Prof. YingLi Tian, and specifically Prof. George Wolberg, who devoted significant time to helpful technical discussion about my research and kept up-to-date with my progress. Further, I would like to thank Prof. Davi Geiger, for his excellent lectures, assisting me in showcasing my work, and attending my thesis presentation.

I thank all my colleagues from the CCNY Robotics Lab, including Mr. William Morris, Mr. Carlos Jaramillo, and Mr. Roberto Valenti, for the academic collaboration and friendships. Similarly, I thank my colleagues at Google’s Project Tango team, including Dr. Johnny Lee, Dr. Joel Hesch, Dr. Esha Nerurkar, and Mr. Simon Lynen, for the technical and professional guidance they provided.
I would like to acknowledge Prof. Maria Tamargo, for the innumerable ways in which she helped me through career advice and personal support.

Finally, I thank my family - for their unquestioning, unwavering support in everything I pursue, and the sacrifices they have made to make this possible.
To Ana, for all your love and support.
Contents

1 Introduction ................................................. 1
  1.1 Motivation .............................................. 1
  1.2 Thesis contribution ................................. 6
  1.3 Previous work ......................................... 8
    1.3.1 Micro-aerial vehicle navigation ................. 8
    1.3.2 3D mapping with depth and RGB-D cameras .... 10
    1.3.3 Data structures for 3D scene reconstruction .. 13

2 Laser-based reconstruction with an MAV .................. 19
  2.1 System architecture .................................... 22
  2.2 State estimation ...................................... 23
    2.2.1 Scan matching ................................... 24
    2.2.2 Scan projection .................................. 26
    2.2.3 Robust height estimation ....................... 28
    2.2.4 Sensor fusion ................................... 31
  2.3 Autonomous flight .................................... 33
    2.3.1 Flight state machine ............................ 33
2.3.2 Control .................................................. 35
2.3.3 Path planning .......................................... 38
2.3.4 Teleoperation ........................................... 39
2.4 Applications .............................................. 40
2.5 Multi-Volume Occupancy Grids ......................... 43
  2.5.1 Volume list representation ......................... 43
  2.5.2 Updating from laser data ......................... 46
  2.5.3 Extracting probabilistic occupancy information ... 51
  2.5.4 Time complexity analysis ......................... 51
  2.5.5 Map overconfidence ............................... 52
2.6 MVOG experiments ........................................ 53
  2.6.1 3D maps ............................................. 53
  2.6.2 Size comparison ..................................... 55
  2.6.3 Time analysis ....................................... 56

3 Scene reconstruction with an RGB-D camera .... 58
  3.1 RGB-D Measurement model ............................. 62
    3.1.1 Preliminaries ...................................... 62
    3.1.2 Depth bias calibration ............................ 63
    3.1.3 Depth uncertainty analysis ...................... 70
    3.1.4 3D distribution .................................. 72
  3.2 Trajectory estimation .................................. 73
    3.2.1 Overview .......................................... 73
    3.2.2 Registration ...................................... 75
    3.2.3 Data association and updating .................... 77
3.3 Place recognition and global alignment .............................................. 78
  3.3.1 Brute-force place recognition .................................................... 80
  3.3.2 Heuristic place recognition ....................................................... 81

3.4 Experiments ..................................................................................... 82

3.5 Application to MAV navigation system .............................................. 88
  3.5.1 System architecture ................................................................. 88
  3.5.2 Sensor fusion ........................................................................... 90
  3.5.3 Control ..................................................................................... 91
  3.5.4 4DOF Path Planning ................................................................. 92
  3.5.5 Experiments ............................................................................. 95

4 Large-scale reconstruction with a mobile device .................................... 97
  4.1 Platform ....................................................................................... 100
    4.1.1 Hardware ............................................................................... 100
    4.1.2 Integrated motion estimation .................................................. 102
    4.1.3 Depth measurement model ..................................................... 103
  4.2 Theory .......................................................................................... 103
    4.2.1 The signed distance function .................................................. 103
    4.2.2 Dynamic truncation distance ................................................... 108
    4.2.3 Space carving ......................................................................... 109
    4.2.4 Colorization ........................................................................... 111
    4.2.5 Discretized data layout ............................................................. 112
    4.2.6 Discretized data updates ........................................................... 114
    4.2.7 Dynamic spatially-hashed volume grid ..................................... 118
    4.2.8 Gradient-based dense alignment .............................................. 119
4.3 Scan integration ................................................................. 122
  4.3.1 Voxel traversal .............................................................. 123
  4.3.2 Voxel projection ............................................................ 126
  4.3.3 Parallelization ............................................................... 128
4.4 Mesh extraction and rendering ............................................... 130
  4.4.1 Online mesh extraction .................................................. 130
  4.4.2 Rendering ................................................................. 133
4.5 Offline processing ............................................................ 133
  4.5.1 Offline pose estimation .................................................. 133
  4.5.2 Multi-dataset reconstruction .......................................... 135
4.6 Experiments ................................................................. 137
  4.6.1 Qualitative experiments ............................................... 137
  4.6.2 Quantitative experiments .............................................. 141

5 Conclusion ................................................................. 146
  5.1 Future work ............................................................... 149

Bibliography ............................................................... 163
## List of Figures

1.1 The CityFlyer MAV ......................................................... 3
1.2 Project Tango mobile devices .............................................. 5

2.1 Visualization of MAV system. ............................................. 20
2.2 MAV system diagram ....................................................... 23
2.3 Laser scan projection ..................................................... 27
2.4 Height reading discontinuities .......................................... 29
2.5 Robust height estimation ................................................ 30
2.6 MAV flight state machine ................................................. 34
2.7 PID controller with velocity feedback .................................. 36
2.8 MAV hovering control experiment ..................................... 37
2.9 MAV waypoint control experiment ..................................... 38
2.10 MAV Path-planning stages .............................................. 39
2.11 Point-based map of corridor ............................................ 41
2.12 2D map of the first floor of Steinman Hall ......................... 42
2.13 3D map of the first floor of Steinman Hall ......................... 43
2.14 MAV trajectory experiment ............................................. 44
2.15 Positive and negative MVOG volume lists ......................... 46
2.16 Process diagram for updating the MVOG map ................................. 47
2.17 MVOG ray rasterization cases .................................................... 49
2.18 MVOG mape stages ................................................................. 54
2.19 2D occupancy grids from an MVOG map .................................. 54
2.20 New College Dataset MVOG map .............................................. 55
2.21 MVOG hallway experiment ....................................................... 56

3.1 3D map from RGB-D data ........................................................... 59
3.2 Depth data distortion ............................................................... 60
3.3 Example depth training image pair .............................................. 65
3.4 Set of depth compensation training images ................................. 66
3.5 Depth compensation results ....................................................... 67
3.6 Compensated depth data view ................................................... 68
3.7 RMS error for Asus camera ....................................................... 69
3.8 Depth uncertainty analysis ......................................................... 72
3.9 RGB-D visual odometry pipeline ............................................... 74
3.10 RGB-D place recognition pipeline ............................................ 80
3.11 Place recognition matrices ...................................................... 82
3.12 Comparison of visual odometry trajectories ............................... 83
3.13 Visual odometry model and data sizes ...................................... 84
3.14 Visual odometry processing duration ........................................ 85
3.15 Visual odometry trajectory with ground truth data ...................... 86
3.16 Large scale map with depth compensation ................................. 87
3.17 CityFlyer MAV with RGB-D camera ......................................... 88
3.18 MAV with RGB-D camera system ............................................. 89
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.19 Velocity estimation from visual odometry</td>
<td>92</td>
</tr>
<tr>
<td>3.20 4DOF path in cluttered environment</td>
<td>94</td>
</tr>
<tr>
<td>3.21 MAV hovering with RGB-D camera</td>
<td>95</td>
</tr>
<tr>
<td>3.22 MAV position control with RGB-D camera</td>
<td>96</td>
</tr>
<tr>
<td>3.23 3D map of room, captured on MAV</td>
<td>96</td>
</tr>
<tr>
<td>4.1 Scene reconstruction for a mobile device</td>
<td>98</td>
</tr>
<tr>
<td>4.2 Overview of the scene reconstruction system</td>
<td>100</td>
</tr>
<tr>
<td>4.3 Project Tango depth images comparison</td>
<td>101</td>
</tr>
<tr>
<td>4.4 Depth training image pair (Project Tango cell phone)</td>
<td>104</td>
</tr>
<tr>
<td>4.5 RMS depth error (Project Tango cell phone)</td>
<td>105</td>
</tr>
<tr>
<td>4.6 Observation ray geometry</td>
<td>107</td>
</tr>
<tr>
<td>4.7 Depth observation probability model</td>
<td>110</td>
</tr>
<tr>
<td>4.8 Memory layout for a single voxel</td>
<td>113</td>
</tr>
<tr>
<td>4.9 Convergence behavior of discrete filters</td>
<td>117</td>
</tr>
<tr>
<td>4.10 Dynamic spatially-hashed volume grid data structure</td>
<td>120</td>
</tr>
<tr>
<td>4.11 Comparison of visual odometry trajectories on Project Tango device</td>
<td>123</td>
</tr>
<tr>
<td>4.12 Reconstructions using dense alignment</td>
<td>124</td>
</tr>
<tr>
<td>4.13 Bipartite volume observation graph</td>
<td>130</td>
</tr>
<tr>
<td>4.14 Reconstruction with offline bundle adjustment</td>
<td>134</td>
</tr>
<tr>
<td>4.15 Multi-dataset reconstruction</td>
<td>136</td>
</tr>
<tr>
<td>4.16 Multi-dataset reconstruction</td>
<td>136</td>
</tr>
<tr>
<td>4.17 Reconstruction comparison using depth-compensated data</td>
<td>138</td>
</tr>
<tr>
<td>4.18 Outdoors reconstruction</td>
<td>139</td>
</tr>
<tr>
<td>4.19 Reconstruction comparison using the different algorithms</td>
<td>140</td>
</tr>
</tbody>
</table>
4.20 Reconstruction using the Freiburg desk public dataset .................. 141
4.21 Affected volume count analysis ................................................. 142
4.22 Comparison of memory consumption ........................................... 144
List of Tables

2.1 Data size analysis .................................................. 57

3.1 Place Recognition Rates ......................................... 82

3.2 Relative Pose Error [meters] .................................... 84

4.1 Comparison of integration algorithms ........................... 128

4.2 Scan integration time [ms] (tablet dataset) ..................... 143

4.3 Scan integration time [ms] (cell phone dataset) ............... 143
Chapter 1

Introduction

1.1 Motivation

Real-time 3D reconstruction is a well-known problem in computer vision and robotics [33]. The task is to extract the true 3D geometry of a real scene from a sequence of noisy sensor readings online. Solutions to this problem are useful for robotic and human-assistive navigation, mapping, object scanning, and more. The problem can be broken down into two components: localization (i.e. estimating the sensor’s pose and trajectory), and mapping (i.e. reconstructing the scene geometry and texture). The mapping portion of the algorithm requires that the sensor is well localized. On the other hand, the localization algorithm will often need to localize against the incrementally-constructed map. Due to the co-dependency between the two components, the problem is sometimes referred to as Simultaneous Localization and Mapping (SLAM).

In this work, we tackle the problem of real-time 3D reconstruction on computationally-constrained platforms, including Micro aerial vehicles (MAVs) and mobile consumer devices
such as cell phones and tablets. The end goal is to be able to create scene reconstructions (or 3D maps) which can serve as a model of the real world and have rich visual and geometric information.

Scene reconstruction on these platforms is a non-trivial problem which presents several notable challenges. First, the devices have limited computational resources, which requires the use of optimized algorithms and clever choices about approximations and tradeoffs. Second, the data provided by the scanning devices can be noisy, biased, or incomplete, requiring careful treatment in order to generate full, consistent models. Finally, even if given perfect data and unlimited computational time, detailed 3D scene reconstruction can consume very large amounts of memory, prohibitive for current generation desktop machines, let alone MAVs or mobile devices. Thus, creating a useful 3D map requires optimized data structures for representing spatial data.

We will primarily target two types of sensors: laser range finders and depth cameras. Both of them provide immediate, metric information about the structure of the scene, albeit with different characteristics. For our quadrotor systems, we will consider both laser range-finders and RGB-D cameras, both of which are small enough that they can be mounted on the platform. For our mobile device system, we will focus exclusively on embedded depth cameras, which have worse performance than their larger, consumer counterparts.

Small laser range-finders have a high range and good accuracy, but return a relatively limited observation of the scene, since they operate in 2D. Unlike laser range-finders, depth cameras return dense 2.5D information in the form of a depth image, but have limited range and much lower accuracy. A depth image contains a measure of the distance from the camera to each observed pixel along the optical axis. Depth cameras can sometimes be combined with a color camera in what is referred to as an RGB-D camera - a device which provides
two concurrent image streams: a conventional color image and a depth image. Together, the two images can be used to obtain a dense, textured 3D model of the observed scene. The properties of RGB-D data, together with the low cost of current devices, have made RGB-D cameras very popular among the computer vision and robotics communities. RGB-D data has been used in various applications, including visual odometry, SLAM, scene modeling, and object recognition.

Micro-aerial vehicles such as quadrotor helicopters (Fig. 1.1), are emerging as popular platforms for unmanned aerial vehicle (UAV) research due to their structural simplicity, small form factor, vertical take-off and landing (VTOL) capability and high maneuverability. They have many military and civilian applications, such as target localization and tracking, 3D mapping, terrain and utility inspection, disaster monitoring, environmental surveillance,
search and rescue, traffic surveillance, deployment of instrumentation, and cinematography.

Numerous research efforts have been made in the field of MAV navigation, ranging from MAV test-bed and flight control design and path planning, to 3D SLAM and multi-robot coordination. As a result, today’s MAVs have improved in autonomy to the level that they can achieve autonomous exploration in structured indoor environments, waypoint following and navigation in open space.

Recently, mobile phone manufacturers have started adding embedded depth and inertial sensors to mobile phones and tablets. In particular, the devices we use in this work, Google’s Project Tango [28] phone and tablet (Fig. 1.2) have very small active infrared projection depth sensors combined with high-performance IMUs and wide field of view cameras. Other devices, such as the Occipital Inc. Structure Sensor [1] have similar capabilities. These devices offer an onboard, fully-integrated sensing platform for 3D mapping and localization, with applications ranging from mobile robots to handheld, wireless augmented reality.

Consider house-scale (300 square meter) real-time 3D mapping and localization on a quadrotor or Tango device. The robot or user moves around a building, scanning the scene. At house-scale, we are only concerned with features with a resolution of about 2-3 cm (walls, floors, furniture, appliances, etc.). We impose the following requirements on the system:

- To facilitate scanning, real-time feedback must be given to the agent. In the case of the mobile device, a visualization must be presented on-screen. In the case of the quadrotor, the visualization may be streamed to a nearby desk station.

- The system must run in real-time, self-contained on device.

- The entire 3D reconstruction must fit inside the device’s limited (2-4GB) memory. Furthermore, the entire 3D model, and not just the current viewport, must be available
at any point. This is in part to aid the feedback requirement by visualizing the model from different perspectives. More importantly, this enables applications like path-planning or augmented reality (AR) to take advantage of the scene geometry for tasks such as collision detection, occlusion-aware rendering, and model interactions.

- The system must be implemented without using GPU resources. The motivating factors for this restriction is that GPU resources might not be available on the specific hardware, such as our MAVs. Even if they are, there are many algorithms already competing for the GPU cycles: in the case of the Tango tablet, feature tracking for pose estimation and structured light pattern-matching for depth image extraction are already using the GPU. We want to leave the rest of the GPU cycles free for user applications built on top of the 3D reconstruction system.
1.2 Thesis contribution

In this work, we will present three separate systems for 3D scene reconstruction. The systems are unified by the desire to produce informative, accurate scene models (maps), under the requirements we imposed. The system mix different platforms (quadrotor vs mobile device) with different sensors (lasers, depth cameras), and thus require different approaches to the problem. Moreover, building the entire functioning system for an autonomous quadrotor or freehand mapping cell phone requires a complex stack of interconnected components. We will provide an overview of the systems as we describe them, and highlight which parts of the system stack we are focusing on solving.

We begin by presenting a laser-based scene reconstruction system for micro-air vehicles in Chapter 2. The system is implemented onboard the CityFlyer quadrotor MAV (Fig. 1.1). We will show a comprehensive system that achieves autonomous flight indoors, including low-level algorithms such as state estimation and control, mid-level (laser-based odometry, obstacle avoidance), and high-level (3D mapping, 3D path-planning, visualization, user interfaces). We will also suggest a novel multi-level occupancy grid (MVOG) data structure for laser-based mapping. Since the main source of data for the system is a laser range-finder scanning the horizontal plane around the quadrotor, some assumptions need to be made about the scene. Namely, we will assume that the environment is mostly rectilinear - the walls and objects have the same vertical profile at different heights, which holds true for typical indoor scenes such as office buildings. This assumption will allow us to reduce many of the 3D problems into 2D ones, and build on top of existing 2D algorithms.

Next, we present a system of scene reconstruction using an RGB-D camera in Chapter 3. Here, we shift focus to the camera calibration, visual odometry and scene reconstruction problems. Since depth cameras provide rich 2.5 information about the scene, we will be
working with true 6Dof camera trajectories and 3D maps. The reconstruction algorithms are designed to be able to work standalone, with no other input than the RGB-D camera. However, at the end of the chapter, we will demonstrate how we can apply the system to our existing CityFlyer quadrotor MAV, by replacing the laser range-finder with the RGB-D camera.

Finally, we present a system for large-scale scene reconstruction on a mobile platform with an integrated depth camera in Chapter 4. We use two different Project Tango devices (Fig. 1.2). In this system, we will focus almost entirely on the data structure and algorithms required for building and visualizing a high-quality map model. 3D mapping algorithms involving occupancy grids [20], keypoint mapping [41] or point clouds [74, 84, 90] already exist for mobile phones at small scale – but at the scales we are interested in, their reconstruction quality is limited. Occupancy grids suffer from aliasing and are memory-dense, while point-based methods cannot reproduce surface or volumetric features of the scene without intensive post-processing. Furthermore, many of the 3D mapping algorithms that use RGB-D data (including the one we present in Chapter 4 are designed to work with sensors like the Microsoft Kinect [50], which produce higher-quality VGA depth images at 30Hz. The density of the depth data allows for high-resolution reconstructions at smaller scales; its high update frequency allows it to serve as the backbone for camera tracking.

In comparison, the depth data from the Project Tango devices is much more limited. For both devices, the data is available at a rate between 3 and 5Hz. This makes real-time pose tracking from depth data a much more challenging problem. Fortunately, Project Tango provides an out-of-the-box solution for trajectory estimation based on a visual-inertial odometry system using the device’s wide-angle monocular camera. We use depth data only to make small optional refinements to that trajectory. This allows us to focus mainly on the
mapping problem.

1.3 Previous work

1.3.1 Micro-aerial vehicle navigation

In recent years, MAVs have been an increasingly popular platform for robotics research. In Chapter 2, we focus on an onboard laser-based navigation system for indoor environments, including pose estimation, SLAM, planning and control. Many researchers have made efforts to develop a MAV capable of performing autonomous navigation tasks in indoor or outdoor environments, using different sets of sensors. In this section, we review previous work which uses onboard sensors without the aid of external systems such as motion capture cameras (for indoor) or GPS (for outdoor).

Roberts et al. [71] used ultrasound and infrared sensors for controlling a custom built quadrotor in a structured obstacle-free environment. The system is intended to use only a minimum set of sensor so only basic tasks such as autonomous take off and landing, altitude and anti-drift control and collision avoidance are performed.

Zhang et al. [97] present a multi-robot system, capable of tracking the pose of a MAV and ground robot. Pose tracking is achieved through vision processing offboard the MAV.

Andersen and Taylor [4] use a UKF and visual information to estimate the pose of the MAV. Data is post-processed on an desktop computer, and no onboard implementation is available.

He et al. [34] presented an indoor navigation system for small-sized quadrotor. The pose estimation is accomplished by an unscented Kalman filter and a path is computed from a pre-defined map. No open source implementation is available and only the attitude control
Ferrick et al. [24] use a Parrot AR.Drone equipped with a small laser rangefinder to construct simplified occupancy grid maps, performing wall following and obstacle avoidance. They do not provide an open-source code and mapping and path planner run on a base station.

Shen et al. [78] present autonomous navigation system using both laser and visual informations for MAV in an indoor environment pursuing the task of multifloor mapping. However they do not provide an open source code.

Blosch et al. [8] present a navigation system in unknown environments based on vision-inferred pose which controls the MAV during the map-building process. However, no obstacle avoidance is accomplished.

The PIXHAWK system proposed by Meier et al. [48] is a MAV provided with a powerful onboard computer able to run both vision based flight control and stereo-vision based obstacle detection in parallel. Their vision based localization exploits an ARK in a test bed arranged with visual markers. Position control, similar to the one proposed in this work, is provided. However, no velocity control is available and no mapping is computed. The entire system is intended to run in a middleware available on-line but not as widely used as ROS.

Grzonka et al. [31] present a comprehensive open-source laser-based navigation system for MAVs in indoor environments. This system handles pose-estimation, mapping, localization and control, with only the device drivers running on-board. Similar approach for position control is adopted. However, since pose estimation and position control run off-board at 10 Hz, its performance is significantly degraded.

Bachrach et al. [5] present an alternative comprehensive system for performing autonomous exploration and map acquisition in indoor environments. Pose estimation is
achieved through a fusion of scan matching and visual odometry. A Rao-Blackwellized particle filter based 2D SLAM is performed and an algorithm for frontier-based autonomous exploration is used. Four independent PID controllers for position and heading provide command for the Low-Level attitude and thrust controllers. SLAM and path planning are not performed on-board and the implementation is not addressed or published under an open-source license.

Achtelik et al. [2] present similar system for vision instead of laser-based navigation. They achieve fast bias-free pose estimation and control of MAV by fusing visual informations from a monocular camera with IMU data and then use this data to control the MAV by a nonlinear dynamic inversion method. They address the problem of navigating both in indoor and outdoor environments, but do not address obstacle avoidance. Their architecture is similar to the architecture presented in this work, including an open-source release of ROS system components. The fundamental difference between the works is that the system we present relies on laser data, while their system relies on a monocular camera.

1.3.2 3D mapping with depth and RGB-D cameras

Mapping paradigms generally fall into one of two categories: landmark-based (or sparse) mapping, and high-resolution dense mapping. While sparse mapping generates a metrically consistent map of landmarks based on key features in the environment, dense mapping globally registers all sensor data into a high-resolution data structure. In this work, we are concerned primarily with dense mapping, which is essential for high quality 3D reconstruction. These are several areas which our of interest to our work: studies of the depth camera measurement model and its calibration, and visual odometry (the process of tracking the potion of a camera from visual data as it moves through its environment).
In the field of camera calibration, Smisek et al. [80] present a calibration procedure for an RGB-D camera, including the intrinsic parameters or the RGB and depth cameras and the extrinsic matrix between them. Further, they examine the relationship between the raw device output and the metric depth, as well as the resolution and quantization of the depth output. They correct for systematic errors in the depth, but the correction is performed using a constant term which is trained only in the 0.7 to 1.3 meter range.

Khoshelham and Elberink [40] discuss the depth accuracy and resolution of an RGB-D camera, including the relationship between the raw depth output and the metric depth. They derive an uncertainty model for the metric depth based on the resolution of the device and the uncertainty of the raw depth. The model considers pixel readings as independent of neighboring pixels.

Park et al. [63] propose a mathematical uncertainty model for 3D visual features. Further, they propose an alternative model from computing the metric depth from the raw depth output.

Olesen et al. [62] propose an uncertainty model for the depth reading based on a parametric model which considers the radial distance of a pixel from the image center and the depth reading at that pixel.

Nguyen et al. [59] propose a depth noise model by measuring lateral and axial measurement distributions as a function of distance and angle of the sensor to a surface.

Herrera et al. [37] present an algorithm to simultaneously calibrate a depth and RGB camera, including their intrinsic parameters and the pose between them. The method is based on observing checkerboards attached to a large flat surface.

In terms of depth image compensation models, our work is most closely related to that of Teichman et al. [85]. In their paper, they describe a procedure for training an unwarping
model based on myopic parameters. Their system uses SLAM-based structure instead of checkerboard images as a source of reference. This has the added advantage of being able to use natural scenes for online calibration. However, it couples the model training with an existing SLAM implementation, which in turn may be already affected by reconstruction errors due to the depth image biases. This fact becomes increasingly important the bigger the uncalibrated biases are. In particular, we demonstrate results from a depth camera on a mobile device with systematic errors large enough to compromise structure-based calibration methods. In contrast, our reference image detection is based solely on the RGB camera measurements and a projective-n-point (PNP) calculation.

In the field of visual odometry and motion estimation, Steinbrucker et al. [82] present a system for frame-to-frame trajectory estimation by minimizing an energy function in the space of the dense depth data. In our previous work [17] we present a system for frame-to-frame registration using edge features, which uses a high-frequency loop for sparse data and a low-frequency loop for dense data.

Henry et al. [35] present a system which uses GPU-accelerated SIFT features and non-GPU FAST features. Images are aligned on a frame-to-frame basis, by using both sparse and dense data. Global refinement is performed offline using Sparse Bundle Adjustment.

Endres et al. [21] present a system which uses sparse SURF, ORB, or GPU-accelerated SIFT descriptors (the choice is configurable). Images are aligned against a subset of previous frames of constant size, in order to increase robustness. The implementation requires multiple threads. Additional refinement of the trajectory can be performed offline.

Newcombe et al. [56] present a system for RGB-D pose tracking and mapping. Their method aligns dense depth data against a model of a surface. The model is augmented with new data. The system requires a GPU-equipped computer.
Kerl et al. [39] describe a RGB-D SLAM system which tracks images using a frame-to-keyframe scheme and performs pose-graph alignment in a separate thread. The system is notable for performing dense alignment between the frames, using data from all pixels. The alignment runs in real time on a desktop CPU; however, it requires more computational time and lower resolution images than our proposed algorithm, making our solution more appropriate for computationally-constrained systems. Furthermore, their work does not demonstrate results from environments larger than a limited office scene; in comparison, we provide results from a larger-scale reconstruction of multiple rooms.

Meilland and Comport [49] present a system which unifies keyframe-based SLAM techniques with volumetric map representations. The system runs in real time, and requires a GPU-equipped computer.

In our previous work [18], we describe a visual odometry pipeline using a persistent feature model. The visual odometry presented in this work (Section 3.2) expands upon our previous results, describing an improved pipeline including a RANSAC step, as well as more experimental evaluations.

1.3.3 Data structures for 3D scene reconstruction

A well established way for creating 2D maps is occupancy grids. Elfes [20] introduced Occupancy Grid Mapping, which divides the world into a voxel grid containing occupancy probabilities. Occupancy grids preserve local structure, and gracefully handle redundant and missing data. While more robust than point clouds, occupancy grids suffer from aliasing, and lack information about surface normals and the interior/exterior of obstacles.

A probabilistic method for integrating readings in occupancy grids can be found in [87]. All cells take on a continuous value between 0 (free) and 1 (occupied), and are initialized
with a value of 0.5. When a sensor reports a certain distance, cells at that distance have their values increased, while cells that lie within the sensor ray area have their values decreased to mark the free space. The probability values are modified according to Bayesian update rules.

A slightly different mapping model for 2D maps is provided by reflection maps [87]. In a reflection map, each cell in the grid keeps track of two counters: one for the number of times a sensor beam was reflected in the cell (hits), and one for the number of times a sensor beam passed through the cell (misses). The ratio

\[
\frac{\text{hits}}{\text{hits} + \text{misses}}
\]

provides a probabilistic measure of the likelihood of a sensor reporting that cell as an obstacle. Maps produced under this model are very close to regular occupancy grids [87].

The direct extension of 2D occupancy grids to 3D are voxel occupancy grids. Work with 3D occupancy grids has been presented in [19], [72], and [51]. While voxel grids support all the algorithms developed for their 2D counterparts, they are often impractical due to their large memory requirements.

An effort to create a more compact data structure is presented in [88] and [70] with their introduction of multi-level surface maps, or MLS maps. Similar to elevation maps ([32], [36]) MLS maps represent 3D structures as height values over a horizontal grid, but allow for the storage of vertically overlapping objects. While this is shown to greatly reduce the memory requirement, MLS maps only record positive sensor data, and provide no mechanism for decreasing the occupancy value of objects located on the map. Thus, any erroneous readings such as false sensor positives are never removed from the map.
Another approach to building 3D maps is by representing the environment using an octree. Rather than storing a fixed-resolution grid, octrees store occupancy data in a spatially organized tree. In typical scenes, octrees reduce the required memory over occupancy grids by orders of magnitude. Work on octree mapping has been done by [47], [93], [26], [23], [64], and [65]. Octomap [94] provides an overview of existing octree approaches, and how they address issues such as updatability, map overconfidence, and compression. Octomap achieves probabilistic, compact 3D maps, but suffer from many of the same problems as occupancy grids: they lack information about the interior and exterior of objects, and are subject to aliasing errors. Further, octrees suffer from logarithmic reading, writing, and iteration times, and have very poor memory locality characteristics.

Curless and Levoy [16] created an alternative to occupancy grids called the Truncated Signed Distance Field (TSDF), which stores a voxelization of the signed distance field of the scene. The TSDF is negative inside obstacles, and positive outside obstacles. The surface is given implicitly as the zero isocontour of the TSDF. While using more memory than occupancy grids, the TSDF creates much higher-quality surface reconstructions by preserving local structure.

Kinect Fusion [57] uses a TSDF to simultaneously extract the pose of a moving depth camera and scene geometry in real-time. Making heavy use of the GPU for scan fusion and rendering, Fusion is capable of creating extremely high-quality, high-resolution surface reconstructions within a small area. However, like occupancy grid mapping, the algorithm relies on a single fixed-size 3D voxel grid, and thus is not suitable for reconstructing very large scenes due to memory constraints. This limitation has generated interest in extending TSDF fusion to larger scenes. Moving window approaches, such as Kintinuous [91] extend Kinect Fusion to larger scenes by storing a moving voxel grid in the GPU. As the camera moves
outside of the grid, areas which are no longer visible are turned into a surface representation. Hence, distance field data is prematurely thrown away to save memory. As we want to save distance field data so it can be used later for post-processing, motion planning, and other applications, a moving window approach is not suitable.

Recent works have focused on extending TSDF fusion to larger scenes by compressing the distance field to avoid storing and iterating over empty space. Many have used hierarchical data structures such as octrees or KD-trees to store the TSDF [96, 12]. However, these structures suffer from high complexity and complications with parallelism.

An approach by Nießner et al. [60] uses a two-layer hierarchal data structure that uses spatial hashing [86] to store the TSDF data. This approach avoids the needless complexity of other hierarchical data structures, boasting $O(1)$ queries, and avoids storing or updating empty space far away from surfaces.

An alternative approach is to represent the environment using point clouds. The map is the aggregation if multiple registered point clouds. Each point is created from a range reading provided by a laser scanner or a stereo-camera [15], [61]. This method have been applied to point clouds generated by depth cameras as well [74, 84, 90, 22]. However, this method does not model free and unknown space, information which is crucial to scene reconstruction under high levels of noise. Thus, point-cloud based methods are suitable for sensors that have very high accuracy. Moreover, the size of the map grows linearly and without an upper bound with the number of sensor readings.

In our work, we use two different data structures. For our MAV system, we suggest a novel multi-level occupancy grid (MVOG) for laser-based mapping. For our mobile platform mapping with depth cameras, we adapt and extend the spatially-hashed data structure of Nießner et al. [60].
The MVOG mapping structure that we propose is closely related to multi-level surface maps and reflection maps. Similarly to MLS, MVOGs group readings into continuous vertical volumes, which are placed over a horizontal grid of fixed resolution. Unlike MLS, however, MVOGs record both positive and negative readings, grouping them into distinct positive and negative volumes. The occupancy information is computed similarly to the reflection map model. Note that although we use the reflection model, which does not directly model occupancy, we informally refer to our maps as “occupancy” maps modeling occupancy probabilities, due to the similarity of the two models. The MVOG data structure is very computationally and memory-efficient, and easily integrates with existing grid-based algorithms for robotics path-finding and obstacle avoidance, but does not support texture information.

For the purposes of scene reconstruction on mobile devices, we consider a different data structure which allows us to extract high-fidelity mesh models, as well as incorporate color readings. By carefully considering what parts of the space should be turned into distance fields at each timestep, we avoid needless computation and memory allocation in areas far away from the sensor.

Because mobile phones typically do not have depth sensors, previous works [84, 58, 22] on dense reconstruction for mobile phones have gone to great lengths to extract depth from a series of registered monocular camera images. Since our work focuses on mobile devices with integrated depth sensors, such as the Google Tango devices [28], we do not need to perform costly monocular stereo as a pre-requisite to dense reconstruction. This allows us to save our memory and CPU budget for the 3D reconstruction itself.

Unlike [60], we do not make use of any general purpose GPU computing. All TSDF fusion is performed on the mobile processor, and the volumetric data structure is stored on the CPU.
Instead of rendering the scene via raycasting, we generate and maintain a polygonal mesh representation, and render the relevant segments of it. Since the depth sensor found on the Tango device is significantly noisier than other commercial depth sensors, we reintroduce space carving [20] from occupancy grid mapping (Section 4.2.3) and dynamic truncation [59] (Section 4.2.2) into the TSDF fusion algorithm to improve reconstruction quality under conditions of high noise. The space carving and truncation algorithms are informed by a parametric noise model trained for the sensor using the method of Nguyen et al. [59].
Chapter 2

Laser-based reconstruction with a micro-aerial vehicle

In this chapter, we present a navigation system for a quadrotor micro aerial vehicle (MAV) and discuss its open-source software implementation (Fig. 2.1). The system, which is designed to work in GPS-denied indoor environments, has been tested with the CityFlyer (Fig. 1.1), a research project at the City College of New York to develop an MAV that is capable of autonomous flight in a variety of three-dimensional spaces, based on the AscTec Pelican quadrotor. All the software discussed runs on-board an in real time on the 1.6Ghz Atom board processor and ARM7 microcontroller.

The first contribution of this chapter is a collection of tools for MAV state estimation. The 9-component state (position, angles, and linear velocities) of the MAV is estimated at a rate of 30Hz, and fused together with IMU readings at a rate of 1Khz using a Kalman Filter. The laser-based framework makes the assumptions that the environment is mostly rectilinear vertically and horizontally, which holds true for typical indoor scenes. The Kalman Filter
fusion framework allows for external sensor updates to be provided, so the system can be extended beyond the laser sensor.

The second contribution of the chapter is a collection of tools for MAV control and path-planning. We assume that the MAV comes with roll, pitch, yaw-rate, and thrust (RPYT) controllers available on the autopilot board. These are typically mapped to the four stick axes of radio remote controllers for quadrotors. We perform modified PID control for the $x$, $y$, $z$ (height), and yaw- (heading) state of the quadrotor, on top of the built-in RPYT control, at a rate of 1Khz. The input to the control can be both position and velocity commands. Additionally, we discuss a 2D costmap-based path-planning algorithm, teleop interfaces, and a state machine providing safe take off and landing behavior. The framework is decoupled and allows for external control commands for a given component. For example,
a user can attach a modified height controller, while keeping the existing $x$-, $y$-, and yaw controllers.

The third contribution of the chapter is a set of interfaces allowing the system to be used with many existing ROS components, including 2D slam, 2D localization, 3D mapping, and visualization. We maintain a strong policy of abstraction and hardware independence. This is accomplished through using existing ROS message and service types, and defining new general MAV messages and services where needed. To ensure compatibility with rest of ROS and future systems, all messages are in SI units, and all frames are in the ENU convention.

The last contribution of the chapter is a set of specific tools related to the AscTec Pelican platform, including a firmware allowing the system to communicate with the AscTec quadrotor driver, 3D visualization models of the AscTec Pelican, and CAD designs for hardware parts.

We provide the open-source implementation of all the tools described, as well as their documentation, online. Where appropriate, we have also provided data sets allowing individual components to be tested.

The chapter is organized in the following way. Section 2.1 presents an overview of the entire system. Section 2.2 describes the state estimation framework, including the laser-based scan matching, altitude estimation, and sensor fusion. Section 2.3 goes over the control and path-planning details. Section 2.4 outlines ways we have used our system in conjunction with existing mapping and SLAM frameworks. Finally, in Section 2.5, we present Multi-volume occupancy grids, and detail experiments we performed with the data structure in Section 2.6.
CHAPTER 2. LASER-BASED RECONSTRUCTION WITH AN MAV

2.1 System architecture

The complete block diagram of our system architecture is shown in Fig. 2.2. The framework is distributed between a ground station, onboard CPU, and onboard microcontrollers. We use the AscTec Pelican system, which has an 1.6 GHz Atom Board processor and 2 ARM7 microcontrollers, of which we utilize one for our firmware.

The ground station is only used for visualization, and to allow a user to teleoperate the MAV (using a joystick or by clicking on the map shown on the screen). Thus, it is a non-critical component.

The onboard CPU handles odometry, mapping, localization, and path-planning. All software running on the CPU operates in the ROS framework. The different components are implemented as nodelets, a ROS mechanism for threaded execution allowing for zero-copy message transport. The flyer interface serves as a generic bridge between ROS and the microcontroller, translating messages and services to packets sent over a serial connection. The implementation of the two-way serial communication is adapted in part from the implementation of [2].

The Autopilot microcontroller handles sensor fusion though a Kalman Filter and control. The implementation (written in C) is provided by the authors. The implementation is minimally coupled with the AscTeC hardware, through a set of functions which translate SI units to the custom units required by the quadrotor drivers, and functions to read and write data from the quadrotor. Therefore, the code should be easily adaptable to other hardware platforms as well.
2.2 State estimation

The state estimation system is presented in detail in Fig. 2.2. The IMU provides readings for the roll ($\phi$), pitch ($\theta$), and yaw ($\psi$) angles of the MAV in the inertial frame, as well as the linear accelerations ($a_x$, $a_y$, $a_z$), measured in the body frame. During the scan projection step, laser data is orthogonally projected onto the horizontal $xy$-plane by using the roll and pitch angles of the last state. The projected data is then passed to a scan matcher. The scan matcher uses the yaw angle reading from the IMU as an initial guess for the orientation of
the MAV. The output of the scan matcher includes $x$, $y$, and yaw angle pose components.

A laser altimeter uses the laser scan readings of the floor, as well as the roll and pitch orientation of the MAV, to estimate the altitude ($z$) of the vehicle. The readings are obtained by using a mirror to deflect a portion of the horizontal laser scanner’s beams downwards. A histogram and threshold filters are applied to detect discontinuities in the floor, assuming the floor is piecewise-linear.

A rough estimate of the linear velocities is computed from the derivative of the position readings, and passed through an alpha-beta filter ($\alpha\beta$F). The estimates from the various components are fused using a Kalman Filter (KF) to produce the final state.

The following subsections describe each step in detail.

### 2.2.1 Scan matching

One of the performance-critical steps used in our state estimation technique is scan matching. An common approach to scan matching is to match each scan to the previous scan, and incrementally compute the global pose. This approach results in fast performance, at the cost of incurring drift errors over time. The scan matching is typically performed using a variation of the Iterative Closest Point (ICP) algorithm [98]. In recent years, significant results have been achieved in the field of real-time 3D point cloud matching. An overview of ICP variants if provided by [75]. A study of the parameters affecting ICP performance is presented in [67].

While scan matchers have been a staple in many robotics localization algorithms, we found that adapting a scan matcher to be used for onboard state estimation is a hard problem. We present the steps taken to reduce the problem to 2D scan matching, and details which allow us to obtain high performance (30hz) and better accuracy using the
limited computational abilities of the MAV CPU.

Our pose estimation relies on the assumption that in indoor environments, obstacles usually appear the same in 2D regardless of the height at which they are observed, due to the rectilinear property. This means that changes in the altitude of the vehicle do not affect the accuracy or output of the scan matcher. We also assume that the environments have sufficient geometric details. The pose estimation will fail when observing:

- environments with highly-nonrectilinear vertical surfaces, such as staircases or inclined walls. Regular furniture or slight wall irregularities do not compromise robustness.
- environments with high degrees of symmetry and no geometric features, such as empty hallways longer than the laser range, or circular empty rooms.

Our implementation is based on on the PL-ICP algorithm [11]. PL-ICP uses a point-to-line metric to solve the iterated corresponding point problem. We provide two extensions to the algorithm, not addressed in the original work.

The first extension is using a constant-velocity model instead of a zero-velocity model during the scan matching. A good way to speed up the convergence of ICP is to provide the algorithm with an initial guess of the change of pose between the scans. Instead of making the common assumption that the robot did not move between the scans, we estimate its instantaneous velocity and use it as a prediction of the current pose. Estimating this velocity vector can be achieved in a naive way by taking the time derivative of the robot’s position. Alternatively, the velocity can be estimated using a Kalman Filter, where the time derivative is fused with acceleration readings from an IMU. If no acceleration readings are available, a simpler filter that can be used is the $\alpha\beta$-filter. This resulted in faster convergence, and consequently, higher update frequencies of the scan matcher.
The second extension is to introduce keyframe scans into the scan matching process. In the classical implementation, each scan is aligned against its immediate preceding scan, and any small error in the scan matching is accumulated over time. Thus, even if the vehicle is relatively stationary, the pose can drift without bound. In our approach, we align each scan against a single previous keyframe scan. The keyframe scan is updated once the vehicle moves a certain distance (for example, 20 degrees or 20 cm.) This approach greatly reduces the overall drift in the estimation. When the vehicle is stationary, the keyframe scan remains the same, and thus error estimation of the position is bounded.

2.2.2 Scan projection

If we know the 3D orientation of the laser scanner, we can reduce the problem to 2D scan matching and perform scan matching on sequences of projected scans instead of raw scans. The two steps required for this are projecting the moving frame, and projecting the laser data. The projected scans and frame further allow us to interface with existing 2D implementations of SLAM, localization, and navigation. An overview of the projection is presented in Fig 2.3. We use a star superscript (*) to denote virtual (orthogonally projected) data and frames.

Let $B$ denote the position of the MAV base in the fixed frame of reference, denoted by world. The transformation between the world frame and the base frame is $T_{\text{base}}^{\text{world}}$, and is the output of the pose estimation system.

We define a virtual MAV base with position $B^*$ and coordinate system ortho, corresponding to the orthogonal projection of $B$ onto the $xy$-world plane. The transformation $T_{\text{world}}^{\text{ortho}}$ between the world and ortho can be obtained by discarding the $z$, $\phi$, and $\theta$ components of $T_{\text{world}}^{\text{base}}$, using the last output available from the pose estimation.

Note that we project the MAV base frame, and not the laser frame itself. This is because
CHAPTER 2. LASER-BASED RECONSTRUCTION WITH AN MAV

Figure 2.3: An overview of the laser projection step, showing the real laser, with frame \textit{laser}, the projected laser, with frame \textit{ortho}, the actual laser data (\textit{scan}), and the projected laser data (\textit{scan}*)

after projection, the laser’s $x$- and $y$- position in respect to the base frame is not constant, but depends on the roll and pitch motion of the MAV. This effect is not present in robots moving in 2D. It becomes more noticeable the further the laser is mounted from the MAV’s center of rotation. To account for this, we express the scan in the coordinates of the vehicle, and not in the coordinates of the sensor.

Let $L$ denote the position of the laser in the \textit{world} frame, and \textit{laser} be the coordinate frame attached to it. In order to perform scan matching with respect to the virtual \textit{ortho} frame, we need to transform the laser scan data from the \textit{laser} frame (attached to the laser scanner) to the \textit{ortho} frame. We begin by expressing the position of the laser in the world
frame:

\[
T_{\text{laser}} = T_{\text{base}}^\text{world} \cdot T_{\text{base}} \tag{2.1}
\]

where \( T_{\text{base}} \) is determined by the vehicle configuration.

The laser readings arrive in the form \( \{r_i, \beta_i\} \), where \( r_i \) are range readings, and \( \beta_i \) are angles at which the readings were taken. The points detected by the laser thus have coordinates

\[
[P_i]_{\text{laser}} = [r_i \cos \beta_i, r_i \sin \beta_i, 0]^T.
\]

Transforming that into the \textit{world} reference frame we get

\[
[P_i]_{\text{world}} = T_{\text{laser}}^\text{world} \cdot [r_i \cos \beta_i, r_i \sin \beta_i, 0]^T \tag{2.2}
\]

We then obtain the orthogonally projected points, first in the \textit{world}, then in the \textit{ortho} frame

\[
[P_i^*]_{\text{world}} = [P_{ix}, P_{iy}, 0]^T \tag{2.3a}
\]

\[
[P_i^*]_{\text{ortho}} = T_{\text{ortho}}^{-1} \cdot [P_i^*]_{\text{world}} \tag{2.3b}
\]

We perform scan registration in respect to the \textit{ortho} frame. The rigid transformation which best aligns two consecutive sets of laser data gives us the instantaneous change in \( x \)-, \( y \)-, and \( \psi \)- pose of \( B^* \) in the \textit{world} frame. This in turn, is equivalent to the instantaneous change in the \( x \)-, \( y \)-, and \( \psi \)- pose of \( B \) in the \textit{world} frame.

### 2.2.3 Robust height estimation

The laser altimeter has been used previously by [5], [30]. We describe it with the addition of a robust filtering technique.

Part of the laser rays from a horizontally-mounted laser scanner are deflected towards
Figure 2.4: Readings from a deflected laser scanner as the vehicle is traveling over an edge. Left (1): readings fall on the floor surface. Middle (2): readings are distributed between floor and elevated surface, with some readings incorrectly interpolated in-between. Right (3): readings fall on the elevated surface.

the ground using a mirror. The measurement to the floor is corrected to account for the roll and pitch angles of the MAV. The laser altimeter assumes that the floor surface is piece-wise linear. If a large measurement discontinuity is detected, the system assumes that there is a discontinuity in the floor - the MAV has moved over a surface with a different height. In this way, we can simultaneously track the vehicle altitude and the floor level.

A problem with this technique is that certain laser scanners, including popular Hokuyo models, may provide an interpolated reading when the laser beam falls on the edge of a discontinuity (see Fig. 2.4). Thus, readings taken of a discontinuous surface get smoothed out to a continuous curve, and thresholding on the measurement jumps fails to detect the object edge properly.

To solve this problem, we deflect multiple laser beams down. Next, we compute a histogram of the $z$ readings, discretization them in a certain bin size. In our experiments, we used 20 beams, and a bin size of 2cm. We next identify the mode of the readings, (the peak of the histogram), and compute the average of all readings belonging to that bin. In this manner, we can robustly identify reading inliers and outliers as the vehicle is moving across an edge on the floor (Fig. 2.4). The results of the height estimation can be seen in Fig. 2.5,
Figure 2.5: Experimental verification of the robust height estimation algorithm. The MAV is flown over several flat objects of varying heights. The algorithm tracks the estimated floor level (blue) and vehicle altitude (red). The vehicle altitude is compared to ground truth data (black) from a motion capture system. (In this experiment, the MAV is flown at an arbitrary height, and does not attempt to maintain a certain altitude.)

which shows the estimated vs ground-truth altitude (from a VICON motion capture system) of the MAV, as well as the estimated floor level, as the vehicle travels forward and backward over several surfaces.

Mounting one mirror pointing down and another one pointing up allows the MAV to estimate the distances to both the ceiling and the floor. This can be accomplished simply by running two instances of the same software described above. The position of the ceiling plane with respect to the MAV can be useful for obstacle avoidance.

We have tested the system using two different laser scanners - a Hokuyo URG-04LX-UG01 and a Hokuyo UTM-30LX. We have released 3D CAD files of the mirror mounting hardware, available online.
2.2.4 Sensor fusion

In order to estimate the state and reduce the amount of noise from the sensor readings we use a cascade of two filters: an Alpha Beta Filter ($\alpha\beta$F) and a Kalman Filter (KF). The $\alpha\beta$F is a simplified case of the Kalman Filter which does not maintain its optimal functionality but has the advantage of being simple and fast since it does not require a system model. The $\alpha\beta$F assumes that the system has two states, where the first state is the integration of the second state (for example, position and velocity). The $\alpha\beta$F is composed of the two typical steps of a Kalman Filter, prediction and correction:

**Prediction:**

\[
\hat{x}_{k|k-1} = \hat{x}_{k-1|k-1} + \Delta T \cdot \hat{v}_{k-1|k-1} \\
\hat{v}_{k|k-1} = \hat{v}_{k-1|k-1}
\]

**Correction:**

\[
r_k = x_k - \hat{x}_{k|k-1}
\]

where $r_k$ is the residual (or innovation) and represents the prediction error calculated as the difference between the new measurement and the estimated position. It is used to correct the two states:

\[
\hat{x}_{k|k} = \hat{x}_{k|k-1} + \alpha \cdot r_k \\
\hat{v}_{k|k} = \hat{v}_{k|k-1} + \frac{\beta}{\Delta T} r_k
\]

where $\alpha, \beta \in [0, 1]$. Lower values $\alpha$ and $\beta$ result in a smoother estimate but with longer delay.

The main reason why the $\alpha\beta$F is used is to obtain a less noisy first estimation of the linear velocity in order to use it as correction in the Kalman Filter, instead of a simple derivative.
of the laser position which increases the amplitude of the noise. The estimation is carried out at 30Hz.

Once we have estimated the position, linear velocities and yaw angle, we pass these values to the Autopilot where they are fused with the IMU readings at a rate of 1Khz. We use a Kalman Filter similar to the one described by [2], where the acceleration readings from the IMU, expressed with respect to the body coordinate frame, are transformed by a rotation matrix to be expressed with respect to the world coordinate frame, and then corrected for gravity. In the filter design all three axis are considered decoupled. In this way, three smaller KFs (one for each axis) perform estimation of position and velocity using prediction by integrating acceleration and correction by position and velocity laser data pre-filtered with the \( \alpha \beta \)F. A fourth KF performs estimation of the yaw angle integrating the high frequency yaw-rate from gyroscope which is then corrected by the yaw angle reading from the laser.

The states, inputs and measurements of the KFs are the following:

\[
\begin{align*}
\mathbf{x}^{(1)} &= [x \ v_x]^T & \mathbf{u}^{(1)} &= [a_x] & \mathbf{z}^{(1)} &= [x_L \ v_{xL}]^T \\
\mathbf{x}^{(2)} &= [y \ v_y]^T & \mathbf{u}^{(2)} &= [a_y] & \mathbf{z}^{(2)} &= [y_L \ v_{yL}]^T \\
\mathbf{x}^{(3)} &= [z \ v_z]^T & \mathbf{u}^{(3)} &= [a_z] & \mathbf{z}^{(3)} &= [z_L \ v_{zL}] \\
\mathbf{x}^{(4)} &= [\psi] & \mathbf{u}^{(4)} &= [\omega_z] & \mathbf{z}^{(4)} &= [\psi_L]
\end{align*}
\]
CHAPTER 2. LASER-BASED RECONSTRUCTION WITH AN MAV

The linear discrete motion model used to describe the system is shown below.

\[ x_k = A_k \cdot x_{k-1} + B_k \cdot u_k \]  
(2.13)

\[ y_k = H \cdot x_k \]  
(2.14)

Where for \( x_1, x_2 \) and \( x_3 \)

\[
A = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \quad B = \begin{bmatrix} \frac{\Delta T^2}{2} \\ \Delta T \end{bmatrix} \quad H = \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\]  
(2.15)

and for \( x_4 \):

\[
A = [1] \quad B = [\Delta T] \quad H = [1]
\]  
(2.16)

2.3 Autonomous flight

2.3.1 Flight state machine

In order to achieve safe and robust flight behavior, we present a state machine which handles the transitions in and out of autonomous flight mode (Fig. 2.6).

The system starts in the Off state, in which the power to the motors is cut off. An engage command advances the state to Engaging, where the Autopilot attempts to communicate with all 4 motor controllers and turn them on. If this step succeeds, the system enters the Idle state, where the motors are powered on and rotating at a minimum idle speed. A similar disengage procedure can transition between the Idle state and the Off state.

Once in the Idle state, a takeoff command enables the autonomous control and advances
the MAV to the *Flying* state. While in that state, the MAV can be in different control modes (Position, Velocity, Direct - described in the next section). A *land* command transitions the state back to *Idle*. There is an intermediate state called *Landing*, during which the MAV is held at the same spot, while the thrust is gradually (over the course of several seconds) decreased linearly from its current value down to 0. Note that this is an open-loop landing behavior, usually initiated once the MAV is within a safe touch-down distance of 10-15 cm above the ground.

If any of the timed transitions fail, or if the vehicle is given an *e-stop* command, the system enters an *Error* state. Power to the motors is immediately shut off, and the MAV must be restarted.
2.3.2 Control

The control system provides 4 different control modes for different purposes. The modes are: Disabled, Direct, Position and Velocity.

Disabled mode disables the high-level control in the Autopilot. This mode is used for manual flight using the Remote Controller (RC) which sends commands directly to the Autopilot low level RPYT controller.

Direct mode bypasses the high-level control allowing users to send commands directly to the Autopilot low-level RPYT controller by an external source. This mode permits users to add their own controller, which can run on the onboard CPU or workstation.

Position mode only enables the position-based high-level control. It can be used for autonomous navigation with the path planner which generates a sequence of waypoints.

Velocity mode only enables the velocity-based high-level control. We perform velocity control separately to the position control in order to be able to switch from one controller to another to send different kinds of commands. For example, we use velocity mode to fly in teleoperation using the joystick sending velocity commands, where the quadrotor does not need to reach a waypoint but just follows the desired velocity, allowing a smooth and constant flight.

We implement position and velocity control separately for each of the $x$, $y$, and $z$ axes, and position control for the yaw axis, Yaw-rate control is already provided by the quadrotor low-level RPYT controller. Therefore, since the axis are decoupled we can choose different modes for each axes; for example, we can have velocity mode for $x$, $y$ and yaw and position for $z$, or direct mode for the first three with $z$ in direct mode etc.

The position controller is based on a cascade structure of two loops, where the outer position loop generates commands to the inner RPYT controller provided by the AscTec
Autopilot. Roll and pitch commands are generated from the outer loop as reference to the inner attitude controller in order to reach and maintain desired $y$ and $x$ position respectively, and thrust commands are generated by the outer controller to control the height by sending desired thrust values. The position controller is based on a modified PID structure (PI-D) as shown in Fig 2.7. The $PI$ block is directly applied to the error as difference of the desired position and current position and its output is added to the output of the $D$ term which is not applied on the derivative of the error but on the negative feedback of velocity value. From a mathematic view, applying the $K_d$ term to the negative velocity is equivalent to applying it to the derivative of the error as long as the waypoint is constant, as shown below.

$$\frac{de(t)}{dt} = \frac{d[x_d - x(t)]}{dt} = \frac{dx_d}{dt} - \frac{dx(t)}{dt} = -v(t) \tag{2.17}$$

During flight, the waypoints are constant for most of the time and change only when the quadrotor reaches the current waypoint. This improves the performances of the controller because it avoids the phenomenon called “derivative kick”, which is the spike due to the derivative of the error during the instantaneous changing of the setpoint value. Furthermore,
the velocity estimation is more accurate and less noisy than a simple derivative. The complete position controller has the following form:

\[
    u(t) = K_p e(t) + K_i \int_0^t e(t) \, dt + K_d [-v(t)] 
\]  

(2.18)

The velocity controller as well as yaw controller are simple PI controllers. Fig. 2.8 shows position plots of the MAV in a hover experiment where it maintains a position around the desired point with oscillations smaller than 20 cm for \(x-\) and \(y-\) axis and around 5 cm for \(z\) for a period close to 4 minutes. Fig. 2.9 shows the quadrotor position as it follows with varying \(x-\), \(y-\), and yaw components.
2.3.3 Path planning

Path planning and obstacle avoidance is performed using a grid-based costmap. An overview of the process is shown in Fig. 2.10. First, a costmap with inflated obstacles is computed, using the map produced by the laser data. A user can click anywhere on the costmap, and a path from the MAV to the goal is computed using Dijkstra’s algorithm. This functionality is provided by the ROS navigation stack.

The second step is to decompose the plan into a sequence of straight-line segments, each of which is collision-free. This is done by a greedy recursive divide-and-conquer algorithm,
Figure 2.10: Path planning stages. Left: A user selects a goal for the MAV. The grid-based costmap and path are generated by the ROS navigation stack. Middle: The plan is decomposed in straight-line, collision-free segments. Right: a sequence of waypoints is generated for the MAV to follow, including rotate-in-place waypoints.

presented in Algorithm 1. The algorithm takes a set of sequential grid cells $P$ (the original plan), and returns a set of grid cells corresponding to end-points of safe, collision-free straight lines.

The third step is to generate a sequence of waypoints from the decomposed plan. Each waypoint contains $[x, y, \theta]^T$ coordinates. Waypoints are generated so that the quadrotor is always traveling with its $x$-axis close to the direction of motion. If the next waypoint heading is similar to the current one (in our implementation, within 20°), then the MAV is allowed to move forward and adjust its heading simultaneously. If a waypoint is at a heading sufficiently different than the current one, an intermediate “turn-on-a-dime” waypoint is generated, which forces the MAV to rotate in place before moving forward.

The straight-line trajectory to the next waypoint is continuously checked for collision in case of dynamic objects, and if collision is detected, the plan is recomputed.

2.3.4 Teleoperation

We provide a generic teleoperation interface for the MAV, allowing the user to send velocity commands, position commands, and state transitions, together with an implementation of
Algorithm 1 Path decomposition algorithm

1: function pathDecomp($P$)  
2:   \( P \) is a set of \((x, y)\) cells. The size of \( P \) is \( N \).  
3:   \( N \leftarrow \|P\| \)  
4:   if collisionFreeLine($P_1$, $P_N$) then return \{\(P_0, P_N\}\}  
5:   else  
6:      \( P^- \leftarrow \{P_0, \ldots, P_{N/2}\} \)  
7:      \( P^+ \leftarrow \{P_{N/2}, \ldots, P_N\} \) return pathDecomp($P^-$) \cup pathDecomp($P^+$)

a teleop client using a PlayStation joystick controller. In our experiments, we observed that the most intuitive control scheme is to use velocity control for the \(x\), \(y\), and \(yaw\) axes, and position control for the \(z\) axis. The user can send velocity commands using the thumb controllers, while the MAV is autonomously kept at a fixed height. Sending zero-velocity commands for \(x\) and \(y\) provides precise stops with very little drift. The resulting effect is the illusion of controlling a holonomic ground MAV which glides around at a certain altitude (also adjustable through the PS3 controller).

The teleop control mode can work in cooperation with the point-and-click method of sending waypoints. At any given time, the user can issue a “hold” command with the joystick controller, resulting in clearing the planner commands, and freezing the vehicle at it’s current position, until further commands arrive. The “hold” command, together with the “e-stop” command, provide a basic safety functionality that we believe should be part of any MAV system with a mixture of autonomous and teleoperated controllers.

2.4 Applications

In the previous sections, we presented experiments demonstrating the state estimation, control, and path-planning capabilities of the system. Next, we briefly present applications
Figure 2.11: A 3D map of a corridor, obtained by flying the MAV along a path of 25 meters, while changing the altitude slightly. A grid with cell size 1 meter is provided for reference.

which use our system in conjunction with existing ROS modules. In all the experiments described, computation was carried out entirely onboard.

The first experiment consists of building a 3D map by adding laser data at poses determined by the state estimation system (Fig 2.11), without the aid of SLAM algorithms. As the figure shows, the state estimation applied to mapping results in locally consistent maps.

In the second experiment, we demonstrate that our state estimation system can easily be used in conjunction with existing mapping tools. The state estimation is used as an odometric input to a 2D SLAM algorithm (gmapping, [29]) running onboard the vehicle in real time. The map is constructed by passing all the projected laser scans to gmapping. The 2D SLAM step performs additional alignment of the data and guarantees loop closure.
CHAPTER 2. LASER-BASED RECONSTRUCTION WITH AN MAV

Figure 2.12: 2D map of the first floor of Steinman Hall at The City College of New York, computed onboard the MAV.

and global consistency. Note that gmapping alone, without the aid of laser-based odometric system, is too slow and cannot solve the problem of 2D SLAM for the MAV. We are able to use it in real time since we provide it with a good odometric estimate from the scan matcher. Also note that the output of gmapping is not used in any critical components such as control or real-time obstacle avoidance. Thus, if it fails, the system would not become unstable. The 3D point cloud is built on top of the 2D-refined data, by commanding the MAV up and down. The results of the experiment can be seen in Fig. 2.12, and Fig. 2.13.

In the third experiment, we use a previously-built map to localize the robot from an unknown position. The localization is performed using the Advanced Monte-Carlo algorithm (AMCL), an implementation of which is provided in ROS.

In the fourth experiment, we perform a fully-autonomous flight over a long trajectory, including narrow doorways. The experiment uses an a priori map of resolution 0.10 meters, also created by the MAV in advance. The MAV starts at a known location, and is given a goal by the operator. Next, the MAV calculates a trajectory and a sequence of waypoints,
and autonomously reaches the goal, as shown in Fig. 2.14. The path-planning and waypoint decomposition took 80 ms. During the flight, the MAV uses AMCL to refine its pose on the map.

Footage of the experiments is shown in the video attachment.

2.5 Multi-Volume Occupancy Grids

2.5.1 Volume list representation

A multi-volume occupancy grid consists of a 2D grid $G$ of square cells $c_{ij}$, $i,j \in \mathbb{Z}$ lying in the $xy$-plane. Any point $\mathbf{p} = [p_x, p_y, p_z]^T$, $\mathbf{p} \in \mathbb{R}^3$ projects onto a cell $c_{ij}$ such that
$i \leq s_{p_x} < i + 1, \ j \leq s_{p_y} < j + 1$. where $s$ is a constant scaling factor between the world and grid coordinates \[88\]. Each cell contains two lists of volumes: $^{+}\mathcal{V}_{ij} = \{ +V_{ij}^0, +V_{ij}^1 \ldots +V_{ij}^n \}$ and $^{-}\mathcal{V}_{ij} = \{ -V_{ij}^0, -V_{ij}^1 \ldots -V_{ij}^m \}$, of sizes $n$ and $m$ respectively. The list $^{+}\mathcal{V}_{ij}$ contains volumes representing positive (obstacle) readings, while $^{-}\mathcal{V}_{ij}$ contains volumes representing negative (free space) readings.

Each volume $V$ is defined using three values: the height of its bottom face $z^{bot}_{V} \in \mathbb{R}$, the height of its top face $z^{top}_{V} \in \mathbb{R}$, and occupancy mass $m_{V} \in \mathbb{R}, m_{V} \geq 0$. We derive a fourth value, the occupancy density $\rho_{V}$. The occupancy mass of a volume corresponds to the amount of sensory information the volume has received. The occupancy density corresponds to the amount of sensory information per unit space.
\[ \rho_V = \frac{m_V}{(z_{v_\text{top}} - z_{v_\text{bot}})A_{c_{ij}}} \]  
(2.19)

where \( A_{c_{ij}} \) is the unit area of the grid cell.

For positive volumes, the occupancy mass comes from sensory information obtained from obstacle readings. For negative volumes, the mass comes from information from free-space readings. All new volumes start off with a density of 1. The occupancy mass of any volume can only increase over time. For example, if we detect a certain point of space as free, we would not decrease the occupancy mass of a positive volume containing the point, but would rather create a negative volume for that region of space. The exact algorithm for manipulating the size and occupancy densities of volumes over time is described in Section 2.5.2.

We impose the additional restrictions on volume lists that:

- Each volume has a height greater than or equal to 1.
- No two volumes in the same volume list overlap.
- The gap between any two volumes in the same list is greater than 1.

The size restriction on minimum volume and gap size is chosen to be 1, which is the same as the resolution restriction in the \( x \) and \( y \) directions. In this way, we guarantee that the minimum vertical resolution the same as our horizontal one. However, the vertical resolution is effectively better than the horizontal one, since volumes are allowed to start and end at non-integer \( z \) values.

Fig. 2.15 shows a possible list combination for grid cell \( c_{0,1} \), displaying the positive (red) and negative (blue) volumes side by side. Note that volumes from the positive list can
overlap with volumes from the negative list, corresponding to the situation that the sensors provided contradicting observations for the same region in space.

### 2.5.2 Updating from laser data

The process of building a MVOG from range readings provided by a laser scanner consists of three steps. First, we rasterize each individual laser reading to obtain which grid cells it crosses. Next, we create and insert new positive and negative volumes into the volume lists of the corresponding grid cells. Last, we examine the modified volume lists, and apply the constraints defined in Section 2.5.1. The process diagram of the update process is shown in Fig. 2.16.
**Rasterization**

An individual laser scan can return either an obstacle reading at a distance $d$ or an out-of-range reading, where $d > d_{\text{max}}$. From the position $\mathbf{L} \in \mathbb{R}^3$ of the laser and the orientation at the time of the scan, we can calculate the end point of the laser ray $\mathbf{L}' \in \mathbb{R}^3$. In this context, an out-of-range reading means that the space between $\mathbf{L}$ and $\mathbf{L}'$ is free, and an obstacle reading carries the same free-space information, with the additional information that $\mathbf{L}'$ is occupied.

By projecting the ray from $\mathbf{L}$ to $\mathbf{L}'$ onto the $xy$-plane, we obtain a list cells $C = \{c_{i_0j_0}, c_{i_1j_1} \ldots c_{i_kj_k}\}$ that the ray crosses, where $C$ is a subset of the grid $G$, $C$ has a length of $k$, and $c_{i_kj_k}$ is the grid cell where the laser ray terminates. We can also calculate the heights where the the laser ray enters the space above each cell $z_{ij}^{\text{enter}}$ and the height where it leaves it, $z_{ij}^{\text{exit}}$ [95]. Note that the laser ray never leaves the last cell $c_{i_kj_k}$, thus we only obtain an entry height value. Instead of an exit height value, we will use the termination height of the laser ray $L'_{z}$. 

![Figure 2.16: Process diagram for updating the MVOG map](image)
Creating new volumes

All new volumes are created with an occupancy density $\rho$ of 1. By knowing the height of the volume and using (2.19), we can instantiate all new volumes with the appropriate occupancy mass $m$.

When adding an out-of-range reading to the map, we create a new volume $V$ for each cell $c_{ij}$ in $C$, apart from the last cell, $c_{ikj}$.

$$z_{V}^{\text{bot}} = \min(z_{ij}^{\text{enter}}, z_{ij}^{\text{exit}}) \quad (2.20a)$$

$$z_{V}^{\text{top}} = \max(z_{ij}^{\text{enter}}, z_{ij}^{\text{exit}}) \quad (2.20b)$$

For the last cell, we have

$$z_{V}^{\text{bot}} = \min(z_{ij}^{\text{enter}}, L'_{z}) \quad (2.21a)$$

$$z_{V}^{\text{top}} = \max(z_{ij}^{\text{enter}}, L'_{z}) \quad (2.21b)$$

Next, we insert the newly created $V$ into the corresponding negative volume list $-V_{ij}$.

When adding an obstacle reading, we repeat the above procedure for each cell in $C$, apart from the last cell, $c_{ikj}$. We need to insert a positive volume of height 1, but depending on the slope of the laser ray, we might need to create an additional negative volume in the same cell. If $\text{abs}(z_{\text{enter}} - L'_{z}) \leq 1$, then we create a volume $V$ such that

$$z_{V}^{\text{bot}} = L'_{z} - 0.5 \quad (2.22a)$$

$$z_{V}^{\text{top}} = L'_{z} + 0.5 \quad (2.22b)$$
Figure 2.17: The two different cases when rasterizing an obstacle reading from a laser. Grid cells in \( C \) are marked in gray. The left diagram shows a rasterization of a laser ray from \([0, 0, 0]^T\) reading an obstacle at \([0, 4.5, 10]^T\), requiring a positive and a negative volume to be inserted in the last cell. The right diagram shows a rasterization of a laser ray from \([0, 0, 0]^T\) reading an obstacle at \([0, 10.5, 4]^T\), requiring only a positive volume to be inserted.

and insert it into the positive volume list \( +V_{ik,jk} \). On the other hand, if \( \text{abs}(z_{\text{enter}} - L'_z) > 1 \), we create and insert \( V \) in the same manner, but also create the additional negative volume \( V' \) to pad the distance between \( z_{\text{enter}} \) and \( V \), and insert it into \( -V_{ik,jk} \). The two cases are illustrated in Fig. 2.17.

Constraint application

The last step in the process is to go through each modified volume list and make sure all the volumes satisfy the constraints we defined in Section 2.5.1. The first constraint refers to the minimum volume size. Any volume \( V \) such that \( z_{V}^{\text{top}} - z_{V}^{\text{bot}} < 1 \) is replaced with a volume \( V^* \)
of height 1 in the following manner:

$$z_{V^*}^{\text{bot}} = \frac{1}{2}(z_{V^*}^{\text{bot}} + z_{V^*}^{\text{top}}) - 0.5$$  \hspace{1cm} (2.23a)$$

$$z_{V^*}^{\text{top}} = \frac{1}{2}(z_{V^*}^{\text{bot}} + z_{V^*}^{\text{top}}) + 0.5$$  \hspace{1cm} (2.23b)$$

Next, we satisfy the constraint that no two volumes in the same volume list overlap by merging them together. Merging two volumes is the main mechanism for incrementally fusing in new sensor information. Two volumes $V^A$ and $V^B$ overlap if $z_{V^B}^{\text{bot}} \in [z_{V^A}^{\text{bot}}, z_{V^A}^{\text{top}}]$ or $z_{V^B}^{\text{top}} \in [z_{V^A}^{\text{bot}}, z_{V^A}^{\text{top}}]$. The resulting volume has an occupancy mass equal to the sum of the occupancy masses of the added volumes. Any two overlapping volumes are replaced by a volume $V^*$ such that

$$V^* = V^A \cup V^B$$  \hspace{1cm} (2.24a)$$

$$m_{V^*} = m_{V^A} + m_{V^B}$$  \hspace{1cm} (2.24b)$$

Last, we satisfy the constraint that the gap between any two volumes is bigger than 1. Two volumes $V^A$ and $V^B$ are too close if $z_{V^B}^{\text{bot}} - z_{V^A}^{\text{top}} \in (0, 1]$. We create a new volume $V^{\text{GAP}}$ corresponding to the gap between $V^A$ and $V^B$, initialized with a density of 1. Then we merge all three volumes into one continuous $V^*$ such that

$$V^* = V^A \cup V^{\text{GAP}} \cup V^B$$  \hspace{1cm} (2.25a)$$

$$m_{V^*} = m_{V^A} + m_{V^{\text{GAP}}} + m_{V^B}$$  \hspace{1cm} (2.25b)$$
2.5.3 Extracting probabilistic occupancy information

MVOGs allow us to extract probabilistic information about the occupancy of each point in space. The two pieces of information that we maintain for any point are the occupancy densities of the positive and negative volumes containing the point, if such volumes exist.

Having defined the positive and negative density functions, we can define the occupancy probability \( p \in [0, 1] \) of a point \( \mathbf{p} = [p_x, p_y, p_z]^T \). Let \( \rho_\mathbf{p}^+ \) be the occupancy density of the positive volume containing \( \mathbf{p} \) (or 0 if no such volume exists). Similarly, let \( \rho_\mathbf{p}^- \) be the occupancy density of the negative volume containing \( \mathbf{p} \). Then, the probability \( p(\mathbf{p}) \) is the ratio between \( \rho_\mathbf{p}^+ \) and the sum of \( \rho_\mathbf{p}^+ \) and \( \rho_\mathbf{p}^- \).

\[
p(\mathbf{p}) = \begin{cases} 
\frac{\rho_\mathbf{p}^+}{\rho_\mathbf{p}^+ + \rho_\mathbf{p}^-} & \text{if } \rho_\mathbf{p}^+ + \rho_\mathbf{p}^- > 0 \\
\text{unknown} & \text{if } \rho_\mathbf{p}^+ + \rho_\mathbf{p}^- = 0
\end{cases}
\]

(2.26)

If both \( \rho^+ \) and \( \rho^- \) in (2.26) are equal to 0, corresponding to the situation when there are neither positive nor negative observations for that point in space, we return an unknown probability.

Using the probability function \( p \), we can construct a standard 2D occupancy grid for any plane in 3D space. The 2D grid can then be used with existing algorithms - for example, localization using a laser that has an arbitrary orientation in space.

2.5.4 Time complexity analysis

Since occupancy volumes are stored in the form of a sorted array, the insertion and lookup operations for volumes are executed in linear time. For a grid cell with lists \( +V_{ij} \) and \( -V_{ij} \)
of length $n$ and $m$ respectively, inserting a positive volume has a runtime of $O(n)$, inserting a negative volume has a runtime of $O(m)$, and the calculation of the probability function $p$ has a runtime of $O(m + n)$.

The linear runtime can be improved by storing the volume lists in a different structure, such as a skip lists, which provides $O(\log(n))$ lookup and constant insertion time [68]. However, we find in practice that the average value of $n + m$ for our test environments remains low. This effectively reduces the runtime of the insertion and lookup operations to constant time.

### 2.5.5 Map overconfidence

When a section of the map has received a large number of negative readings, it normally takes the same amount of positive readings to raise the probability ratio to 0.5 (and vice versa). Thus, maps can become overconfident. This can lead to serious problems, especially when there are dynamic obstacles in the environment. For example, if the robot has recorded a hallway as free multiple times, and a person suddenly walks in, the robot will not be able to correct the section on the map that the person is occupying until a large number of sensor readings have been collected.

To overcome this issue, occupancy grids that use the Bayesian update model in conjunction with log-odds propose clamping, which effectively sets a limit on the number of readings it takes to change the probability value of a cell in the situation described above [94].

MVOGs overcome the issue of map overconfidence by introducing a decay factor in the map. Both positive and negative volume masses are periodically multiplied by a constant factor $k_d$, smaller than 1. Decaying the map has the effect that new sensor readings have a higher weight than previous sensor readings. Note that for map areas that do not receive new
sensor readings, the occupancy probability remains the same before and after the update step, since both the negative and positive volume masses are scaled down equally.

\section*{2.6 MVOG experiments}

\subsection*{2.6.1 3D maps}

We tested the performance of MVOGs using two experiments.

For the first experiment, we use readings from a Swissranger 4000 depth-camera mounted on a quadrotor MAV. Since the depth-camera is too heavy to be carried in flight, we maneuver the MAV by manually carrying it indoors, obtaining multiple views of an office room. The room has two desks and a person sitting on a chair in front of a computer, as well as various clutter. The 6 DoF pose of the UAV is determined by a combination of inertial readings and scan matching \[52\].

The resulting map is shown in Fig. 2.18. The left image shows the raw point cloud data. The middle image shows the positive and negative volumes. The right image shows areas of the map where the probability of occupancy is greater than 0.5 (obstacles).

In Fig. 2.19, we show a cross-section of the map taken at 2 different heights. For comparison, we also show a high-resolution occupancy grid created by a 2D particle-based SLAM algorithm \[29\] of the same room, using independent data from a laser scanner.

In the second experiment, we build an MVOG using a subset of the New College Dataset \[81\], collected by a ground robot outdoors. We use the first 17500 laser scan readings, corresponding to the first loop around the Epoch A campus. We use the raw odometry data provided from the robot.

Fig. 2.20 shows the resulting MVOG map using different resolutions, as well as a cross-
CHAPTER 2. LASER-BASED RECONSTRUCTION WITH AN MAV

Figure 2.18: Stages in the generation of a 3D MVOG map of an office room. The left image shows the raw point cloud data. The middle image shows the positive (red) and negative (blue) volumes. The right image shows areas of the map where the probability of occupancy is greater than 0.5 (obstacles). The volumes in the right image are colored by height.

Figure 2.19: Occupancy grids obtained from the MVOG in Fig. 2.18 by taking a cross-section at different heights. The left image shows the occupancy at 1.6 meters above the floor. The entire room is navigable. The middle image shows the occupancy at 1.5 meters, where the top of a person’s head and room clutter are visible. The right image provides a high-resolution 2D map of the room for comparison. The 2D map was created by an independent laser data and a particle SLAM algorithm.

section of the map taken at the height of the laser scanners. Note that the double walls and incorrect geometry of the loop are a result of the drift in the odometric pose estimation provided in the dataset.
Figure 2.20: MVOG of the New College Dataset (Epoch A, one loop). The left image shows 0.10m resolution. The middle image shows 0.50m resolution. The right image shows the occupancy map obtained by taking a cross-section of the 0.10m MVOG, taken at the height of the laser scanners. Note that the double walls and incorrect geometry of the loop are a result of the drift in the odometric pose estimation provided in the dataset.

We perform an additional experiment to demonstrate the updatability of the MVOG in the presence of dynamic obstacles. In that experiment, we collect readings from a Swissranger 4000 depth camera positioned statically. The camera observes a hallway while a person walks in and out of view (Fig. 2.21).

All experiments were performed using ROS (Robot Operating System) [69] as an underlying architecture to transmit messages between various components of the system.

2.6.2 Size comparison

In this section, we analyze the memory consumption of the MVOGs, compared to raw point cloud, voxel grid, and Octomap representations. We have made at effort at a fair benchmarking, even though we are comparing inherently different data structures. The memory sizes provided for all four data structures assume single-precision floating point representations. For voxel grids, the reported size is for the minimum 3D grid that accommodates all the data. For MVOGs, the reported size is for the minimum 2D grid that accommodates
Figure 2.21: Experiment demonstrating the addition and removal of dynamic objects to the map. The Swissranger scans an empty hallway (left image), a person walks into the frame (middle image), and the person has walked out of view (right image).

the data, and no restriction on the vertical bounds. For Octomap, we report the size of the pruned (lossless compression) tree needed to represent the data. Out-of-range sensor readings present in the experiments are disregarded in all four representations.

The results from the indoor and outdoor data sets, mapped at different resolutions, are summarized in Table 2.1.

### 2.6.3 Time analysis

The insertion time for the 0.10\textit{m} resolution office room experiment was on average 0.122\textit{s} per Swissranger scan, where each scan contained an average of 24510 range readings. This results in an average insertion rate of 200778 readings per second. The insertion time for the 0.10\textit{m} resolution outdoor dataset was on average 0.008\textit{s} per 2D laser scan, where each scan contained 180 range readings. This results in an average insertion rate of 21886 readings per second. Note that this includes a high number of out-of-range (50\textit{m}) readings in every
### Table 2.1: Data size analysis

<table>
<thead>
<tr>
<th></th>
<th>Point cloud</th>
<th>Voxel grid</th>
<th>Octomap</th>
<th>MVOG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>New College</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.00m res.</td>
<td>27.430 MB</td>
<td>0.422 MB</td>
<td>0.189 MB</td>
<td>0.150 MB</td>
</tr>
<tr>
<td>0.50m res.</td>
<td>27.430 MB</td>
<td>3.245 MB</td>
<td>0.785 MB</td>
<td>0.588 MB</td>
</tr>
<tr>
<td>0.10m res.</td>
<td>27.430 MB</td>
<td>390.750 MB</td>
<td>56.309 MB</td>
<td>16.438 MB</td>
</tr>
<tr>
<td><strong>Office room</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.10m res.</td>
<td>16.823 MB</td>
<td>0.259 MB</td>
<td>0.273 MB</td>
<td>0.080 MB</td>
</tr>
<tr>
<td>0.05m res.</td>
<td>16.823 MB</td>
<td>2.031 MB</td>
<td>1.597 MB</td>
<td>0.328 MB</td>
</tr>
<tr>
<td>0.02m res.</td>
<td>16.823 MB</td>
<td>30.175 MB</td>
<td>15.877 MB</td>
<td>2.730 MB</td>
</tr>
</tbody>
</table>

scan. Timing was carried out on a standard desktop CPU (dual Intel Xeon Processor E5504, 2.0GHz).
Chapter 3

Scene reconstruction with an RGB-D camera

In this chapter, we present our work on 3D mapping of indoor environments with an RGB-D sensor. An example of a 3D reconstruction produced by our system is shown in Fig. 3.1. Our research has been motivated from the perspective of mobile robotic perception; therefore, there is a strong focus throughout our work on computational efficiency and real-time performance. The three key problems we address are the calibration and measurement model of the camera, the trajectory estimation of a moving camera, and place recognition from RGB-D images for detecting and dealing with trajectories revisiting the same location multiple times. We believe our contribution to the first two areas are applicable beyond the field of robotic perception, to anyone working on indoor modeling with RGB-D data. Our place recognition work is primarily included for the sake of system completeness and as a straightforward method for offline post-processing; we don’t claim any significant contribution over more established methods in the field.
We begin by examining the accuracy and precision of depth images produced by RGB-D cameras. Accuracy refers to the closeness of the measurements to the truth, while precision refers to the variation of repeated measurements under the same conditions. Compared to their more expensive range-sensing counterparts such as laser scanners, RGB-D cameras have lower precision and accuracy. Consider Fig. 3.2, which shows the result of a simple experiment where we placed an RGB-D camera 4 meters away from a flat wall. The point cloud visible in the figure is the result of aggregating multiple depth readings. An ideal sensor would produce a thin, straight line as the top view of the point cloud. It is immediately obvious that the real camera has not only a high uncertainty, but also a high degree of systematic error. More importantly, the systematic error varies across different sections of the image, producing geometric distortions.

We propose a per-pixel polynomial model calibration technique to remove the bias in the depth readings. The calibration technique relies on observing a flat checkerboard at many different locations. A “ground truth” pose is determined by detecting the checkerboard in the RGB image. With this model, incoming depth images can be unwarped so that the bias in each pixel is removed. We demonstrate experimentally that unwarping the data has
CHAPTER 3. SCENE RECONSTRUCTION WITH AN RGB-D CAMERA

Figure 3.2: Top-down view of a point cloud obtained by measuring a flat wall using an RGB-D camera. The point cloud is constructed by averaging multiple depth images to remove the effect of random noise. The curvature in the point cloud indicates that the depth readings ($Z$) for each pixel have a strong systematic error.

a significant impact on the performance on visual odometry and mapping algorithms that rely on depth readings. We further formulate an uncertainty model for the depth reading in each pixel. The uncertainty is based on a Gaussian mixture model of depth readings of the pixels in a local neighborhood window. When treating each 3D point as a probabilistic distribution, the calibration technique accurately determines the mean, while the uncertainty model predicts the standard deviation.

The probabilistic measurement model forms the basis of a visual odometry pipeline for trajectory estimation. We begin by computing the locations of sparse features in the in-
coming RGB-D image, and their corresponding 3D coordinates in the camera frame. Next, we align these features against a global model dataset of 3D features, expressed in the fixed coordinate frame. After calculating the transformation, the model is augmented with the new data. We associate features from the RGB-D image with features in the model, and update them using a Kalman filter framework. Any features from the image which cannot be associated are inserted as new landmarks in the model set.

The model (which starts out empty) gradually grows in size as new data is accumulated. To guarantee constant-time performance, we place an upper bound on the number of points the model can contain. Once the model grows beyond that size, the oldest features are dropped to free up space for new ones. By performing alignment against a persistent model instead of only the last frame, we are able to achieve significant decrease in the drift of the pose estimation. We perform the trajectory estimation in real time, at rates of 30Hz or higher (the camera outputs the images at 30Hz, but our algorithm is able to process them faster). It uses a single thread, and does not require a GPU.

This chapter is organized as follows. Section 3.1 describes RGB-D camera measurement model, including the depth image calibration procedure and noise models. Section 3.2 presents the trajectory estimation pipeline, focusing on the visual odometry algorithm. Section 3.3 presents our work on place recognition from RGB-D images, and its application to global keyframe-based pose alignment. Finally, we present our experimental results using multiple datasets in section 3.4.
3.1 RGB-D Measurement model

3.1.1 Preliminaries

In the context of this work, we define an RGB-D image as an image with 3 standard color channels and an additional depth channel. The depth channel contains a measure, in meters, of the distance from the camera optical center to the scene, along the camera optical axis. We adopt the standard camera coordinate frame convention with the $z$ axis pointing along the optical axis. We expect RGB-D images to have complete color information, and incomplete, yet very dense, depth information.

An RGB-D camera will typically output a color and depth image produced by two separate cameras, RGB and infrared, each with its own intrinsic parameters and distortion coefficients. The device and its software driver might have a built-in calibration for the intrinsics of the two cameras, as well as the extrinsic pose between them. This allows for the two images to be properly undistorted, and for the depth image to be reprojected into the frame of the RGB camera to form a single unified RGB-D image.

In the case when the factory calibration is not provided, or is not sufficient, a user can perform a custom calibration in two steps. First, the intrinsic parameters of the RGB and infra-red (IR) cameras are determined, including a radial distortion model. This is performed by observing a black and white checkerboard of known size in multiple images. For this procedure we used existing toolboxes implemented with OpenCV [9]. The approach has been well documented, for example in [89]. A point of interest is that the infrared pattern projected by the RGB-D camera interferes with the corner detection on the IR images of the checkerboard. Therefore, it is necessary to place something in front of the IR projector which diffuses the infrared light pattern (a white sheet of paper sufficed in our experiments).
This allows the treatment of the IR image as a standard monochrome image.

The second step is determining the extrinsic matrix between the IR and RGB cameras. This can be performed by treating the two cameras as a stereo pair, and calibrating by using a checkerboard observed in multiple images. Stereo camera calibration is also a well studied problem. We use an implementation based on the OpenCV stereo calibration functions [9]. Treating the two cameras as a stereo pair can be achieved by the diffusion of the IR image described above.

Once the RGB-D image is properly formed, we can represent the image as a dense, 3D point cloud. Given a pixel $q = [U, V, Z]^T$, where $U$ and $V$ are the image coordinates, and $Z$ is the measured depth, in meters, we can express $q$ as a 3D point $p = [X, Y, Z]^T$, in the camera coordinate frame:

$$
X = \frac{Z}{f_x}(U - c_x) \quad (3.1a)
$$

$$
Y = \frac{Z}{f_y}(V - c_y) \quad (3.1b)
$$

where $f_x, f_y$ are the focal lengths, and $c_x, c_y$ are optical centers of the RGB-D image, taken from the RGB-D intrinsic matrix. We treat $U, V, Z$ as random variables. The former two reflect the uncertainty location in sparse features such as corners. The latter reflects the error in the sensor’s depth measurements.

### 3.1.2 Depth bias calibration

Once we have obtained the intrinsic and extrinsic camera matrices, we can use the RGB camera as a source of reference for calibrating the depth camera. While not a real ground-truth measure, we assume that the measurement of the checkerboard in the RGB image
is significantly more accurate than the depth data returned in the depth image. We note that the results of any subsequent depth calibration are subject to the quality of the extrinsic, intrinsic, and distortion model calibration of the cameras, as well as the checkerboard detection on the RGB image.

For performing the depth calibration, we use a large checkerboard printed on a flat surface. We record RGB and depth image pairs of the checkerboard at various distances to the camera and locations in the image. Since there is a limit on how large we can print the checkerboard on a flat surface, when we move the checkerboard further away from the camera it does not cover the entire field of view of the RGB image. Therefore we need to shift the checkerboard and create an image mosaic. Alternatively, if a significantly large, flat wall is available, one could attach the checkerboard to the wall and observe it from various distances.

To mitigate the effects of the noise in the depth readings, the depth image used is the result of averaging 1000 consecutive depth images of the same static scene. An RGB and (average) depth image together form a training pair. Fig. 3.3 shows an example of two training pairs.

In each training pair, we detect the checkerboard corners in the RGB image and calculate their corresponding 3D coordinates in the RGB camera frame. Next, we transform them into the IR camera frame using the extrinsic matrix between the cameras. Once the corners are expressed in the IR frame, we compute the equation of the plane which defines the checkerboard surface using the projective-n-points (PNP) algorithm. We generate a ground-truth (reference) depth image such that any pixel that belongs within the quadrilateral defined by the outermost four corners is given a depth value belonging to the plane surface. Any pixels which are outside the checkerboard are not used. Thus, a training image pair only
allows for calibration in a small area of the image, which gets smaller as the checkerboard is placed at greater distances. Effectively, this means that we need to record a large number of training pairs in order to cover the entire field of view. Fig. 3.4 shows an overview of the data we collected. On the left is a side view of multiple reference checkerboards. In the middle are the corresponding measured depth readings for each checkerboard. On the right is a closeup for the reference and measured locations for a single checkerboard, clearly showing the discrepancy between the two. Note that for clarity, the image only shows 50 checkerboard images, while our calibration used 173.

Once all $K$ training images are collected, we create a set of training points for each checkerboard pixel in each image. We denote the measured distance at a pixel $q = [u, v]$ as $z_{uv}^{(k)}$, and the corresponding reference distance $Z_{uv}^{(k)}$. Since the same pixel $q$ will be observed at multiple images, we use $k$ to express the index of the training image. We define a “corrected”
reading $\tilde{z}_{uv}$ described by a set of $c_{0uv}$, $c_{1uv}$ and $c_{2uv}$ coefficients:

$$\tilde{z}_{uv} = c_{0uv} + c_{1uv}z_{uv} + c_{2uv}z_{uv}^2$$  \hspace{1cm} (3.2)

The total error function for a given pixel location $u, v$ can be defined as

$$e(u, v) = \sum_k \left( z_{uv}^{(k)} - \tilde{z}_{uv}^{(k)} \right)^2$$  \hspace{1cm} (3.3)

where $k$ iterates across all the images where the pixel $u, v$ was inside the checkerboard area.

We wish to find the per-pixel coefficients $c_{0uv}$, $c_{1uv}$ and $c_{2uv}$ which minimize the error for
Figure 3.5: Depth calibration results for two different pixels in the image ([522, 126] and [103, 30]), obtained from training points at varying depths. The figure shows the polynomial fit (solid line) and its deviation from the ideal sensor (dashed line). One of the pixels systematically overestimates the depth $Z$, while the other pixel systematically underestimates it.

We accomplish this by fitting a second-degree polynomial to the data.

$$\arg \min_{c_{0-2uv}} e(u, v)$$

We accomplish this by fitting a second-degree polynomial to the data.

Fig. 3.5 displays the result of the fit for two different pixels in the image. The figure demonstrates that some parts of the depth image consistently overestimate the depth reading, while others consistently underestimate it. Since the fit is performed for each pixel, the end result is $640 \times 480$ sets of coefficient triplets. In our implementation, we store them as three VGA images $I_{c_0}, I_{c_1}, I_{c_2}$, each for the respective coefficient ranks.

Once the coefficient images are recovered, they can be applied to incoming depth images.
Figure 3.6: Top-down view of point cloud created by observing a flat wall with furniture in front of it. The image shows the point cloud before and after applying the polynomial correction. The unwarp behavior is especially visible at the lower-left corner of the image.

to “unwarp” them and remove the systematic bias in the depth reading on a per-pixel basis. The results of the unwarping for a single depth image are displayed in Fig. 3.6. The figure shows the point clouds reconstructed from the original depth image and the unwarped one. We also performed RMS-error analysis of the error between the measured distance $z$ in the depth image and reference $Z$ obtained from the RGB checkerboard image. Fig. 3.7 presents the results for an Asus Xtion-PRO sensor. We define the RMS errors for a given the uncalibrated and calibrated image $k$ as:

$$
ed_{rms}^{(k)} = \left( \frac{1}{n} \sum_{u,v} (Z_{uv}^{(k)} - z_{uv}^{(k)})^2 \right)^{\frac{1}{2}}$$

(3.5a)

$$
\tilde{e}_{rms}^{(k)} = \left( \frac{1}{n} \sum_{u,v} (Z_{uv}^{(k)} - \tilde{z}_{uv}^{(k)})^2 \right)^{\frac{1}{2}}
$$

(3.5b)

where $u$ and $v$ iterate over all the checkerboard pixels observed in that image. Each data
Figure 3.7: RMS error in $Z$ before calibration $e_{rms}^{(k)}$ and after $\tilde{e}_{rms}^{(k)}$, for Asus Xtion-PRO camera. Each data point represents the error over the checkerboard pixels in the $k$-the checkerboard image. Multiple checkerboard images at varying distances $Z$ from the camera are used.

A point in the figure represents the RMS error of $z$ over all the pixels in a given checkerboard test image, versus the average depth of the checkerboard. The results with and without calibration are displayed. The figure shows that the RMS error is significantly lower when the polynomial unwarping is performed on the depth images, and the error improvement becomes more pronounced as the distance between the object and the camera grows.
3.1.3 Depth uncertainty analysis

Khoshelham and Elberink [40] present the following formulation for the uncertainty in $Z$:

$$
\sigma_z = \frac{1}{fb} \sigma_d \mu_z^2
$$

(3.6)

where $f$ is the effective IR camera focal length, and $b$ is the baseline distance between the IR camera and IR projector. After calibration, the authors in [40] obtain the following expression:

$$
\sigma_z = 1.45 \times 10^{-3} \mu_z^2
$$

(3.7)

where $\mu_z$ and $\sigma_z$ are expressed in meters.

We extend the model in two ways. First, the locations of features detected by a sparse feature detector are subject to error; therefore, we allow for uncertainty in the $U$ and $V$ variables. Second, we assume that the depth uncertainty is dependent not only on the depth readings of a given pixel, but also on its neighbors in a local window. We will show that using these assumptions, we can predict the magnitude of the depth uncertainty better than the previously published model.

Let us begin by assuming that $U$ and $V$ are independent random variables distributed according to a normal distribution $\mathcal{N}(\mu_u, \sigma_u)$ and $\mathcal{N}(\mu_v, \sigma_v)$ respectively. Further, let $\sigma_u$ and $\sigma_v$ inform the following approximate Gaussian kernel of size $3 \times 3$:

$$
W = \frac{1}{16} \begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1
\end{bmatrix}
$$

(3.8)
Assuming that $Z$ is normally distributed, we can define a random variable $\hat{Z}$, which is a mixture of the $Z$ variables in a local window $\{i : \mu_u - 1 \leq i \leq \mu_u + 1\}$ and $\{j : \mu_v - 1 \leq j \leq \mu_v + 1\}$. The weights of the mixture $w_{ij}$ are chosen according to the kernel $W$. The mean and variance of the resulting Gaussian mixture $\hat{Z}$ are

\[
\hat{\mu}_z = \sum_{i,j} w_{ij} (\mu_{z_{ij}}) 
\]
\[
\hat{\sigma}_z^2 = \sum_{i,j} w_{ij} \left( \sigma_{z_{ij}}^2 + \mu_{z_{ij}}^2 \right) - \hat{\mu}_z^2
\]

At this stage, we have two alternative models for the uncertainty: $\sigma_z$, estimated according to the simple model in equation 3.6, and $\hat{\sigma}_z$, estimated according to the Gaussian mixture model. To evaluate which model has more predictive power, we gather $n$ depth images of a static scene. For each pixel in the image, we calculate the uncertainty $\bar{\sigma}_z$ in the metric depth $z$ according to

\[
\bar{\mu}_z = \frac{1}{n} \sum_{m=1}^{n} z_m 
\]
\[
\bar{\sigma}_z^2 = \frac{1}{n-1} \sum_{m=1}^{n} \left( \bar{\mu}_z - z_m \right)^2
\]

We call this the observed uncertainty. When $n$ is large (we used 200 images) we can assume that the observed uncertainty approaches the true uncertainty for the RGB-D camera measurement. Next, we take a single depth image, and generate the two predicted uncertainties $\sigma_z$ and $\hat{\sigma}_z$. Fig. 3.8 shows a comparison between the observed and predicted uncertainties according to both models. We note that the Gaussian mixture model predicts the uncertainty much better than the simple model, especially around the edges of objects. This comes from
Figure 3.8: Uncertainty analysis for the depth readings of an RGB-D camera. Left: RGB image, shown for visualization only. Right: observed (ground truth) uncertainty ($\bar{\sigma}_z$), obtained by taking 200 depth images of a static scene and calculating the mean and standard deviation on a per-pixel basis. Color is scaled as the log of the uncertainty. Center-left: uncertainty predicted from a single depth image, according to the simple model ($\sigma_z$). Center-right: uncertainty predicted from the same depth image, according to the Gaussian mixture model ($\tilde{\sigma}_z$). We demonstrate the Gaussian model uncertainty predicts the true uncertainty more accurately, especially around object edges.

the fact that the RGB-D camera produces readings for edge pixels, which tend to jump from foreground to background.

### 3.1.4 3D distribution

We can estimate the 3D uncertainty $\Sigma$ of a point $p$ from Eq. 3.1.

$$
\Sigma = \begin{bmatrix}
\sigma_x^2 & \sigma_{xy} & \sigma_{xz} \\
\sigma_{yx} & \sigma_y^2 & \sigma_{yz} \\
\sigma_{zx} & \sigma_{zy} & \sigma_z^2
\end{bmatrix}
$$

(3.11)
where

\[
\begin{align*}
\sigma_x^2 &= \frac{\sigma_z^2(\mu_u - c_x)(\mu_v - c_y) + \sigma_u^2(\mu_z^2 + \sigma_z^2)}{f_x^2} \\
\sigma_y^2 &= \frac{\sigma_z^2(\mu_u - c_x)(\mu_v - c_y) + \sigma_v^2(\mu_z^2 + \sigma_z^2)}{f_y^2} \\
\sigma_{xz} &= \sigma_{zx} = \frac{\sigma_z^2(\mu_u - c_x)}{f_x} \\
\sigma_{yz} &= \sigma_{zy} = \frac{\sigma_z^2(\mu_v - c_y)}{f_y} \\
\sigma_{xy} &= \sigma_{yx} = \frac{\sigma_z^2(\mu_u - c_x)(\mu_v - c_y)}{f_x f_y}
\end{align*}
\]

The above expressions are derived with \( \mu_z \) and \( \sigma_z \) from the simple depth uncertainty model. We can then approximate \( \Sigma \) in terms of the Gaussian mixture model by replacing \( \mu_z \) and \( \sigma_z \) with \( \hat{\mu}_z \) and \( \hat{\sigma}_z \) respectively. We approximate \( p \) as a multivariate Gaussian distribution with mean \( \mu = [\mu_x, \mu_y, \mu_z]^T \) and covariance \( \Sigma \).

### 3.2 Trajectory estimation

#### 3.2.1 Overview

The entire visual odometry pipeline is shown in Fig. 3.9. The trajectory estimation begins with extracting sparse features in each incoming RGB-D image \( I_t \). The features are detected on the intensity channel of the RGB image. We have experimented with several choices of feature detectors, including SURF [6], ORB [73], and Shi-Tomasi [79] keypoints. While our implementation offers a configurable choice between them, we found that the Shi-Tomasi features offer the best trade-off between robustness and computational speed.

Next, we estimate the 3D normal distribution for each feature, according to the uncer-
Figure 3.9: Pipeline for the trajectory estimation. We align sparse feature data from the current RGB-D frame to a persistent model. The data is represented by 3D points with covariance matrices.

tainty equations defined in Sec. 3.1. From this, we generate a set of 3D features \( D_t = \{ d_i \} \). Each feature \( d = \{ \mu^{[D]}, \Sigma^{[D]} \} \) has a mean and covariance matrix. The \( D \) set is expressed in the camera reference frame.

We have a similar model set of 3D features, \( M_t = \{ m_j \} \) with \( m = \{ \mu^{[M]}, \Sigma^{[M]} \} \) expressed in the fixed frame of reference. The \( D \) set needs to be aligned to the \( M \) set. This happens in 2 steps. First, we align \( D_t \) to \( D_{t-1} \) using a 3-point RANSAC [25] transformation estimation. The feature correspondences between the two frames are established using feature matching with ORB descriptors, computed on the Shi-Tomasi corners. This step eliminates any outlier
features. Next, we align $D_t$ to the previous model $M_{t-1}$, using the transformation estimated by RANSAC as a prediction to do Iterative Closest Points (ICP) \cite{seitz2006}. Note that we do not use the associations generated by RANSAC to do ICP, but choose to compute them all over iteratively. This is because the associations from RANSAC are only between the current and last frame, while in the ICP step we are interested in aligning the 3D features to the entire model, which potentially contains features not seen in the previous frame. While this might seem like a duplicated effort, in practice the ICP algorithm converges quickly in only several iterations since it was initialized with a RANSAC transformation.

The choice of ORB features is motivated by their computational efficiency. Since the ICP matching is the core of our algorithm, and the feature-based matching serves only as an initial prediction, the choice of descriptor and the feature-matching accuracy has a diminished effect on the end result, as long as enough feature matches are present to start the ICP optimization in its basin of convergence.

Once the final transformation is found, we transform the set $D_t$ into the fixed frame (expressed as $D'_t$). Next, we establish the final correspondences between the two sets $D'_t$ and $M_{t-1}$. Re-observed features in the model are updated using a Kalman filter, and new features are added to the previous model, resulting in the new model $M_t$. The rest of this section describes the steps in more detail.

### 3.2.2 Registration

We define a distance function $dist$ which measures the distance between two features $f_a$, $f_b$, normally distributed with means $\mu_a$ and $\mu_b$ and covariance matrices $\Sigma_a$ and $\Sigma_b$.

$$dist(f_a, f_b) = \sqrt{\Delta_{f_a f_b} (\Sigma_a + \Sigma_b)^{-1} \Delta_{f_a f_b}^T} \quad (3.12)$$
where
\[
\Delta_{a,b} = \mu_a - \mu_b
\] (3.13)

The distance function is based on the Mahalanobis distance from a point to a distribution.

We use ICP to align the data set \(D\) to the model set \(M\). As an initial guess for the ICP registration, we use a RANSAC transformation estimation between the previous data features and the new data features.

The ICP algorithm has 2 steps of interest: generating correspondences between the two input sets, and calculating the transformation which minimizes the distance between the correspondences. In the classical ICP formulation, the correspondences are generated using nearest neighbors in Euclidean space; the transformation is also estimated by minimizing the sum of squared Euclidean distances.

We use a modified ICP algorithm, in which we establish approximate correspondences using the Mahalanobis distance. First, we build a kd-tree [54] of the model \(M\), by using the means of the features. Next, for each feature \(d\) in the data \(D\) we find the \(k\) nearest Euclidean neighbors from \(M\). Finally, we iterate through all \(k\) candidates, and find the one which has the smallest Mahalanobis distance. This allows us to leverage the efficiency of kd-trees, which cannot be directly used with non-linear functions such as the Mahalanobis distance. In our implementation, we use a small size for \(k\) (for example, 4).

The rest of the ICP algorithm remains the same. We note that while it is possible to optimize a Mahalanobis distance as the objective function for the best transform, we do not do so, and this is a possible area of improvement. An example of an algorithm which implements a similar optimization (albeit in the context of dense data) is Generalized ICP [77].
3.2.3 Data association and updating

We begin the data association step by rotating the data set $D$ into the fixed frame of reference, and refer to it as $D'$. Let the current transformation between the fixed and camera coordinate frames is $T$, consisting of a rotation and translation:

$$T = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}$$  \hspace{1cm} (3.14)

We can transform the mean vector and covariance matrix according to:

$$\mu' = R\mu + t$$  \hspace{1cm} (3.15a)

$$\Sigma' = R\Sigma R^T$$  \hspace{1cm} (3.15b)

Next, for each point $d'_i$ in the transformed $Data$ set, we find the approximate nearest Mahalanobis neighbor $m_j$ in the $Model$.

$$dist(d'_i, m) = \sqrt{\Delta_{d'_i m} (\Sigma[M] + \Sigma[D'])^{-1} \Delta_{d'_i m}}$$

We consider two points to be associated if the distance between them is lower than a threshold $\epsilon$. Typical values for $\epsilon$ include 7.82 or 11.35. The two thresholds correspond to the 95% and 99% probability tests that the data point is sampled from the given model distribution.

Any features in $D'$ which cannot be associated are inserted as new members in $M$. The model is bounded in size, so if the maximum allowed size is exceeded, we remove the oldest features in the model. This is achieved by using a ring-buffer implementation.
For each feature which is associated, we perform a Kalman filter update. We treat the distributions \( m \) as the prior, and the distribution \( d \) as the observation.

The predicted distribution is the same as the prior state of the model at time \( t - 1 \).

\[
\tilde{\mu}_t = \mu_{t-1}^{[M]} \\
\tilde{\Sigma}_t = \Sigma_{t-1}^{[M]}
\] (3.16a, 3.16b)

The following are the update (correction) equations for the new state at time \( t \):

\[
K_t = \tilde{\Sigma}_t \left( \tilde{\Sigma}_t + \Sigma_t^{[D]} \right)^{-1} \\
\mu_t^{[M]} = \tilde{\mu}_t + K_t \left( \mu_t^{[D]} - \tilde{\mu}_t \right) \\
\Sigma_t^{[M]} = (I - K_t) \tilde{\Sigma}_t
\] (3.17a, 3.17b, 3.17c)

### 3.3 Place recognition and global alignment

We perform global alignment using a keyframe-based pose-graph approach. New keyframes are generated heuristically, using either distance or overlap metric. The distance metric triggers a new keyframe when the linear or angular distance traveled between the current camera pose (as reported by the visual odometry) and the camera pose of the last keyframe exceeds a certain threshold (for example, 0.3m or 30 degrees). The overlap metric triggers a new keyframe when the number of features in the data set \( D \) which have a correspondence in the model \( M \) falls under a certain threshold.

Each of the RGB-D keyframes informs a vertex in a graph. The vertex is described by the keyframe’s 6-DoF pose. Edges in the graph are described by the relative pose between two
keyframes. The visual odometry pipeline provides the edge information for time-consecutive keyframes. To perform large-scale loop closure and global alignment, we need to detect pairs of non-consecutive keyframes and a relative pose between them. Once the graph is built, we optimize for the pose of all keyframes using a non-linear graph optimizer \((g^2o, [43])\). Currently, the pose graph adjustment does not modify the location of the features in the model used for trajectory estimation. However, since we perform the alignment as the final step at the end of the reconstruction run, this is not an issue.

The main problem in the approach described above is finding pairs of RGB-D keyframes that observe the same scene, so we can calculate their relative pose. We refer to this as the \textit{place recognition} problem. Our general approach to the place recognition problem is the following: select a pair of images, try to align their 3D features using a sample-and-consensus algorithm, and mark them as correlated if the algorithm finds a transformation model with a large number of inlier features. We present two ways to implement this approach: a \textit{brute-force} implementation, which is deterministic, and considers all possible image pairs and transformation models, and a \textit{heuristic} implementation, which uses non-deterministic, randomized algorithms to find corresponding images and transformations. The former approach is prohibitively computationally expensive, but it’s \textit{complete}, so we use it as a bench-mark to validate the performance of our heuristic approach, which is computationally efficient.

The generalized form of both approaches can be seen in Fig. 3.10. We have a set of keyframes \(K\) with size \(k\). First, we create a binary candidacy matrix \(Q\) of size \(k \times k\). For any element \(Q_{ij} = 1\) we perform a pairwise matching test between \(K_i\) and \(K_j\). The number of corresponding matches is stored as \(C_{ij}\), forming the correspondence matrix \(C\). Finally, we apply a threshold of minimum correspondence count on \(C\) to obtain the binary association matrix \(A\). The details of how the correspondence matrix \(C\) is created and what the pairwise
matching test consists of are described below.

### 3.3.1 Brute-force place recognition

In the brute force place recognition algorithm, the candidacy matrix $Q$ has 1 in every entry. For the pairwise matching test, we developed an algorithm called *ExSAC*, or Exhaustive Sample And Consensus. ExSAC is a deterministic version of RANSAC which considers every possible set of (fixed-size) samples to find the best model. In our case, the minimum sample size to determine the 6DoF transformation is 3 features. Thus, we use 3-point ExSAC, which considers all combinations of 3-point correspondences between a pair of RGB-D images. For each 3-point set, a rigid transformation is computed, and the number of features which are inliers to this model are counted. The output of the algorithm is the size of the best model, and the corresponding transformation. If we account for the symmetry of the test, and exclude reflexive testing, the pairwise matching is performed in total $(k - 1)^2/2$ times, or $O(k^2)$. The size of the best model is stored in the correspondence count matrix $C$, which is
then thresholded to obtain the association matrix \( A \).

### 3.3.2 Heuristic place recognition

In the heuristic place recognition algorithm, we first compute a match-count matrix \( M \). An entry \( M_{ij} \) corresponds to the number of features in \( K_i \) which have their top nearest neighbors (in feature descriptor space) present in \( K_j \). Next, for each index \( i \), we consider the \( N \) top-scoring keyframes, in terms of match count, and mark them as 1 in the candidacy matrix \( Q \). The result is that \( Q \) has at most \( Nk \) candidates. For each candidate, we apply the 3-point RANSAC test to determine the number of correspondences. From there, the algorithm is the same as its brute-force counterpart.

The advantage of the heuristic test comes from the fact that the correspondence test needs to be performed \( O(kN) \) times, with \( N \ll k \). Furthermore, since the correspondence test uses random sampling instead of exhaustive sampling, we can use much fewer iterations to find a model which approximates the best optimal model. The main free parameter in the heuristic approach is \( N \), or the number of top candidates to consider.

The results for a dataset of 250 keyframes can be seen in Fig. 3.11. On the left is the correspondence count matrix \( C \) for the brute force test, and the respective association matrix \( A \). On the right is the candidacy matrix \( Q \) using the heuristic approach, and the respective association matrix \( A \). The heuristic association matrix is nearly identical to the brute-force association matrix, validating the approach. We further analyze the recognition rate of the heuristic approach, using the brute-force as a baseline for comparison. We define the recognition rate as

\[
1 - \frac{\text{false negatives}}{\text{total associations}}
\]  

(3.18)
where “Total associations” is the number of keyframe pair associations established by the brute force approach, and “False negatives” is the number of associations which were missed by the heuristic approach. The results for different number of candidates $N$ are shown in Table 3.1.

### 3.4 Experiments

In the first experiment, we perform a qualitative evaluation of our visual odometry pipeline with RGB-D data recorded in an indoor environment with no ground-truth data. The camera is moved along a loop, and placed back at its starting point. We replicate the data 5

<table>
<thead>
<tr>
<th>$N$ candidates</th>
<th>False negatives</th>
<th>Recognition rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>169 out of 887</td>
<td>0.809</td>
</tr>
<tr>
<td>10</td>
<td>35 out of 887</td>
<td>0.961</td>
</tr>
<tr>
<td>15</td>
<td>26 out of 887</td>
<td>0.971</td>
</tr>
<tr>
<td>20</td>
<td>19 out of 887</td>
<td>0.979</td>
</tr>
</tbody>
</table>
Figure 3.12: Comparison of trajectory estimation with persistent model (left) vs frame-to-frame ICP. Top row: side views, $xz$-plane. Bottom row: top view, $xy$– plane. The trajectory shown consists of 5 repeated loops, with approximately 2000 images processed in each loop.

Fig. 3.12 shows the trajectories generated using our persistent model pipeline (left), versus trajectories generated by frame-to-frame ICP (right). We show that our approach is able to correctly solve small loops, keeping the trajectory error bounded. Fig. 3.13 shows the respective sizes of the model and data feature sets. The model set grows in size during first loop, while the environment is observed for the first time. In subsequent loops, very few new features are added to the model, since most of them are correctly reobserved and associated.
In the second set of experiments, we evaluated the accuracy of the trajectory estimation using publicly-available RGB-D datasets with ground-truth trajectory information from a motion-capture camera system \[83\]. Table 3.2 shows the RPE (Relative Pose Error) for a number of different trajectories in indoor settings. We have chosen the RPE metric in order to evaluate the effectiveness of our visual odometry system. In the experiment evaluating the RPE error, we do not use our place-recognition or pose-graph-based alignment. The

Table 3.2: Relative Pose Error [meters]

<table>
<thead>
<tr>
<th>Dataset</th>
<th>RGBD-SLAM</th>
<th>ccny_rgbd</th>
</tr>
</thead>
<tbody>
<tr>
<td>freiburg1_360</td>
<td>0.089</td>
<td>0.121</td>
</tr>
<tr>
<td>freiburg1_desk</td>
<td>0.033</td>
<td>0.043</td>
</tr>
<tr>
<td>freiburg1_desk2</td>
<td>0.054</td>
<td>0.067</td>
</tr>
<tr>
<td>freiburg1_room</td>
<td>0.095</td>
<td>0.068</td>
</tr>
<tr>
<td>freiburg1_rpy</td>
<td>0.046</td>
<td>0.064</td>
</tr>
<tr>
<td>freiburg1_xyz</td>
<td>0.021</td>
<td>0.030</td>
</tr>
<tr>
<td>freiburg2_desk</td>
<td>0.017</td>
<td>0.023</td>
</tr>
<tr>
<td>freiburg2_rpy</td>
<td>0.010</td>
<td>0.023</td>
</tr>
<tr>
<td>freiburg2_xyz</td>
<td>0.006</td>
<td>0.008</td>
</tr>
<tr>
<td>freiburg3_long_office_household</td>
<td>n/a</td>
<td>0.028</td>
</tr>
</tbody>
</table>
Table compares error of our implementation (ccny-rgbd) with an existing implementation of [21] (RGBD-SLAM). The results show that our implementation performs slightly worse on most datasets (approximately 1cm more error), significantly better on one dataset (3cm less error), and significantly worse on one dataset (3cm more error).

When comparing the results, we emphasize that our implementation runs in real time (30Hz) on VGA data using a single core of a desktop computer. Moreover, the processing time for each image is on average 22ms, and almost never exceeds 33ms, resulting in minimal latency (see Fig. 3.14). We were able to obtain similar results on an Atom 1.6 Ghz processor using QVGA resolutions. Using the same setup and default parametrization for the RGBD-SLAM implementation, we were able to obtain frequencies of 5 to 10Hz on the desktop machine, and unable to run in real time on the Atom computer. Thus, we believe our pipeline offers a very computationally-efficient solution at a small cost of accuracy. This trade-off is
especially important for systems which require real-time perception such as Micro-air vehicle flight which motivate our research. Fig. 3.15 shows the trajectory of the visual odometry (vo) pipeline for the freiburg2.desk dataset. The figure also shows the trajectory after the pose-graph optimization, which was performed offline at the end of the experiment.

In the third experiment, we evaluated the effectiveness of the entire system, including the depth calibration, trajectory estimation, and pose-graph-based global alignment. We performed a large-scale indoor mapping experiment. We used an Asus Xtion-PRO camera carried by hand, exploring three rooms and returning to the original position in the first room. The results of the experiment can be seen in Fig. 3.1.

Finally, we explore how depth bias calibration affects the different modules in the system.
We repeated the three-room experiment, with and without unwarping the depth images using our calibration model. The results of the experiment can be seen in Fig. 3.16. On the left is the final map result (top-down orthographic projection) with the uncalibrated data. On the right is the result using the unwarped (calibrated) data. The figure demonstrates that the unwarped data produces significantly better results, even with offline graph-based optimization performed in both cases. The systematic skew in the walls of the map in the uncalibrated experiment can be attributed to the slight concavity of the uncalibrated depth data when observing flat surfaces, resulting in incorrect trajectory estimation.
3.5 Application to MAV navigation system

3.5.1 System architecture

To perform autonomous flight using the RGB-D system, we modified the CityFlyer, replacing the laser range-finder with an Asus Xtion Pro Live camera (see Fig. 3.17). Similarly to before, the quadrotor was also equipped with a 1.86 GHz Core2Duo processor with 4GB of RAM and a Flight Control Unit (FCU) board with 2 ARM7 microcontrollers, an Inertial Measurement Unit (IMU) and a pressure sensor. The system architecture is shown in Fig. 3.18.

We use one of the two microcontrollers, the so-called High Level Processor (HLP), to run our custom firmware that handles sensor fusion and control, while the Low Level Processor (LLP) is responsible for attitude control, IMU data fusion and hardware communication. The powerful on-board computer is able to manage visual odometry, 3D SLAM and motion planning. The entire framework is distributed between a ground station, the on-board
computer and the FCU. The ground station is only used for visualization and teleoperation.

Our framework uses ROS [69] as middleware, allowing communication among all the different components of the software (implemented as nodelets, a ROS mechanism for zero-copy message transport). The HLP and the onboard computer communicate with each other through the serial interface, where the Flyer Interface sends ROS messages and services translated into packets. Communication between the two ARM7 microcontrollers (HLP and LLP) of the FCU board is via an I²C bus.

Figure 3.18: MAV with RGB-D camera system diagram. External contributions are marked in red.
3.5.2 Sensor fusion

The output pose of the visual odometry is sent through the serial interface to the HLP, where it is fused with IMU data at a rate of 1 Khz. The high-frequency Kalman filter’s output are fed into the controller, enabling stable flight. As in Section 2.2.4, we cascade an Alpha Beta Filter \((\alpha\beta F)\) and a Kalman Filter (KF). The \((\alpha\beta F)\) runs on the on-board computer and provides a smoother evaluation of its input data (without an actual probabilistic analysis). We use it to reduce the noise in the first estimation of the velocity, which is a simple derivative of the visual odometry position data. Hence, the output of the \((\alpha\beta F)\) is sent over serial interface and serves as correction in the KF. In this indoor application we assume that the quadrotor moves with low velocity and following quasi-rectilinear path and in-place rotations. This assumption allows us to decouple the axis in the KF design and to compute the linear acceleration relative to the fixed frame, from the IMU reading, as explained in [2]. We use three smaller Kalman filters for each position axis as well as a Kalman filter for yaw.

The discrete state space linear model of the KF is:

\[
\begin{align*}
x_k &= A_k \cdot x_{k-1} + B_k \cdot u_k \quad (3.19a) \\
z_k &= H_k \cdot x_k \quad (3.19b)
\end{align*}
\]

where the state, input and measurement for \(x\) (and similarly \(y\) and \(z\)) are:

\[
\begin{align*}
x &= [x \; v_x]^T \\
u &= [a_x]^T \\
z &= [x_{vo} \; v_{xvo}]^T
\end{align*}
\]

while for yaw we have:

\[
\begin{align*}
x &= [\psi]^T \\
u &= [\omega_z]^T \\
z &= [\psi_{vo}]
\end{align*}
\]
where \( a \) is the linear acceleration detected by the IMU, expressed with respect to the fixed frame and \( \omega \), the angular velocity. The matrices \( A, B \) and \( H \) of the system in (3.19) for \( x \), \( y \) and \( z \) are:

\[
A = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} \frac{\Delta T^2}{2} \\ \Delta T \end{bmatrix}, \quad H = \begin{bmatrix} 1 \\ 1 \end{bmatrix}
\]

(3.22)

and for \( \text{yaw} \):

\[
A = [1], \quad B = [\Delta T], \quad H = [1]
\]

(3.23)

Fig. 3.19 shows the output result of the KF for the \( x \)-component of the linear velocity.

### 3.5.3 Control

The control system provides position and velocity control separately for each axis. It is based on a cascade structure of two loops, where the inner loop is provided by the Low level Controller (LLC) implemented in the LLP. It controls roll, pitch, yaw-rate and thrust (RPYT). The outer controller generates RPYT commands to the inner loop controller. Roll and pitch commands are generated from the outer loop \( x-y \)-controllers as reference to the inner attitude controller. Similarly, thrust and yaw rate commands are generated from the height and yaw controller, respectively. The position controller is based on a modified PID, while velocity and yaw controller are PI and P controllers, respectively, as explained in [18].
3.5.4 4DOF Path Planning

This section introduces a quadrotor path planner in $x$, $y$, $z$ and yaw directions. This implementation is based on a search approach where the state space is discretized using a state lattice of motion primitives [66] and an incremental and anytime version of the A* algorithm with Euclidean distance heuristic. This module has been tested in simulation in real-time in combination with the rest of the systems presented in this chapter, on an identical computer as the CPU onboard the MAV.

State space discretization

The quadrotor state space is discretized following a state lattice, a graph search space that integrates motion planning constraints within state exploration. In this case, the state space is four-dimensional, combining the quadrotor position in Euclidean space $(x, y, z)$ with the yaw orientation $\psi$. State space exploration is executed following a set of motion primitives.
Motion primitives are short, kinematically feasible path segments, that can be combined together to produce longer and more complex paths. Any combination of motion primitives yield a path that complies to the non-holonomic constraints imposed by the motion planning problem. Motion primitives are pre-computed, and their traversal cost is multiplied by a user selected weight to obtain the motion cost. Weights are assigned to each motion primitive, in order to model preferences of one primitive over the others, e.g., penalizing changes in altitude while moving forward, in order to keep next positions centered within sensors field of view. Collision checking is performed online while exploring the search graph.

Planning results are greatly affected by motion primitives selection, in terms of planner times, planner completeness, and resulting path quality. The planner can not obtain a feasible path if it cannot be produced by a combination of available motion primitives. For example, backwards paths cannot be generated if backwards motion primitives are not pre-computed and made available in the set. A richer set of motion primitives improves state space coverage adding flexibility to the planner, but there is a trade-off in computation time, as each new motion primitive increases the branching factor at each state.

**Search algorithm**

The described state lattice is explored using a graph search algorithm. This algorithm is a variant of the A* search extended with anytime and incremental capabilities called ARA* (Anytime Repairing A*) [44]. ARA* anytime capability is obtained by executing a series of A* searches where the heuristic is inflated by a factor $\epsilon > 1$, and reducing this factor on each execution. With an inflated heuristic, A* search gives more relevance to the heuristic estimation. This results in a faster algorithm by means of losing optimality, but it has been shown that the computed path sub-optimality is bounded to $\epsilon$ times the cost of the optimal
Figure 3.20: Four-dimensional path (blue) in a cluttered indoor environment. Path starts from actual quadrotor pose (left reference frame) to a user selected goal pose (right reference frame). Intermediate quadrotor poses are shown along the path (colored arrows).

solution [44]. ARA* starts with a high $\epsilon$ value in order to obtain a feasible path very fast. If time is available, $\epsilon$ is decreased and a search is executed again reusing computation from previous search. If enough time is available to reduce $\epsilon$ to 1, the heuristic is not inflated anymore, and the last search returns the optimal solution.

The motion primitives used in our implementation complies with a state lattice discretization of 0.25m per cell of the 3D Euclidean space, and $\pi/4$rad for yaw orientation $\theta$. A typical path query takes 283ms average in a single core (maximum time allowed is 500ms) until the optimal path is obtained, in an indoor environment 30x30x5m in size at 0.25m resolution. For larger environments, more motion primitives, or finer space resolution, obstacle free paths can still be obtained before reaching the optimal path ($\epsilon = 1$) within the
CHAPTER 3. SCENE RECONSTRUCTION WITH AN RGB-D CAMERA

Figure 3.21: Control performance in a hovering experiment over 100s. 3D view (left) and top view (right)

500ms time budget. An example of the 4D path obtained in a cluttered indoor environment is shown in Fig. 3.20.

3.5.5 Experiments

To evaluate the functionality of our system, we performed several experiments in autonomous flight where waypoints were sent through an off-board workstation. In the first experiment shown in Fig. 3.21, we commanded the quadrotor to hover in place for a time of 100 seconds. In the experiment of Fig. 3.22, the quadrotor changed its $x$ and $y$ position after sending a sequence of waypoints. Both experiments prove the effectiveness of state estimation and control with a maximum error of 20 cm or less. Fig. 3.23 demonstrates the 3D SLAM capability of the system. The quadrotor flew autonomously in a large room, with all the computation carried out on-board. The 3D SLAM algorithm receives pose data from the visual odometry and generates a sequence of RGB-D keyframes. The SLAM algorithm tests association between the incoming keyframe and the previous ones to provide correction of the quadrotor trajectory while building a 3D map.
CHAPTER 3. SCENE RECONSTRUCTION WITH AN RGB-D CAMERA

Figure 3.22: Position control with a sequence of waypoints, varying $x$ and $y$.

Figure 3.23: 3D map of a room captured during autonomous flight on the CityFlyer quadrotor MAV.

(a) Point cloud  
(b) Octomap
Chapter 4

Large-scale scene reconstruction with a mobile device

We present an overview of our entire system in Fig. 4.2. We discuss each section component in detail, beginning with Section 4.1, which describes the platform’s hardware, sensors, and built-in motion tracking capabilities.

In Section 4.2, we discuss the theory of 3D reconstruction using a truncated signed distance field (TSDF), introduced by [16], and used by many state-of-the-art real-time 3D reconstruction algorithms [57, 91, 92, 10, 60]. The TSDF stores a discretized estimate of the distance to the nearest surface in the scene. While allowing for very high-quality reconstructions, the TSDF is very memory-intensive. The size of the TSDF needed to reconstruct a large scene may be on the order of several to tens of gigabytes, which is beyond the capabilities of current mobile devices. We make use of a two-level data structure based on the work of Nießner et al. [60]. The structure consists of coarse 3D volumes that are dynamically allocated and stored in a hash table; each volume contains a finer, fixed-size voxel grid which
stores the TSDF values. We minimize the memory footprint of the data by using integers instead of real numbers to do the TSDF filtering. Our main contributions here begin by how we adapt the TSDF fusion algorithms, including dynamic truncation distances and space carving techniques, in order to improve reconstruction quality from the noisy data. We further describe how to use the sensor model in order to switch to the integer representation with the lowest possible data loss from discretization. We discuss how this data discretization impacts the behavior and convergence of the TSDF filter. Finally, we present a dense alignment algorithm which uses the gradient information stored in the TSDF grid to correct for drift in the built-in pose estimation.

In Section 4.3, we present two methods for updating the data structure from incoming depth data: voxel traversal and voxel projection. While each of the algorithms has been described before, we analyze their applicability to the different data formats that we have (mobile phone vs tablet). Furthermore, we discuss how we can leverage the two-tier data structure to efficiently parallelize the update algorithms in order to take advantage of multicore CPU processing.
In Section 4.4, we discuss how we extract and visualize the mesh. We use an incremental version of Marching Cubes [45], adapted to operate only on relevant areas of the volumetric data structure. Marching Cubes is a well-studied algorithm; similar to before, our key contribution lies in describing how to use the data structure for parallelizing the extraction problem, and techniques for minimizing the amount of recalculations needed. We also discuss efficient mesh rendering. Some of our bigger mesh reconstructions (Fig. 4.14) reach close to 1 million vertices and 2 million faces, and we found that attempting to directly render meshes of this size quickly overwhelms the mobile device, leading to overheating and staggered performance.

In Section 4.5, we describe offline post-processing which can be carried out on-device to improve the quality of the reconstruction for very large environments where the our system is unable to handle all the accumulated pose drift. Furthermore, we present our work on extending scene reconstruction beyond a single dataset collection trial. A reconstruction created from a single trial is limited by the device's battery life. The operator might not be able to get coverage of the entire scene in a single trial, requiring to revisit parts later. Thus, we present a method for fusing multiple datasets in order to create a single reconstruction as an on-device post-processing step. The datasets can be collected independently from different starting locations.

In Section 4.6, we present qualitative and quantitative results on publicly available RGB-D datasets [83], and on datasets collected in real-time from two devices. We compare different approaches for creating and storing the TSDF in terms of memory efficiency, speed, and reconstruction quality.
Figure 4.2: Overview of the scene reconstruction system. Components contributed by our work are highlighted in blue.

4.1 Platform

4.1.1 Hardware

We designed our system to work with two different platforms - the Project Tango cell phone and tablet (Fig. 1.2). The cell phone has a Qualcomm Snapdragon CPU and 2GB of RAM. The tablet has an NVIDIA Tegra K1 CPU with 4GB of RAM.

The two devices have a very similar set of integrated sensors. The sensors are listed in Fig. 4.2, left. The devices have a low-cost inertial measurement unit (IMU), consisting
of a tri-axis accelerometer and tri-axis gyroscope, producing inertial data at 120Hz. Next, the devices have a wide-angle (close to 180°), monochrome, global-shutter camera. The camera’s wide field of view makes it suitable for robust feature tracking. The IMU and wide-angle camera form the basis of the built-in motion tracking and localization capabilities of the Tango devices. We do not use the IMU data or this camera’s images directly in our system.

Next, the devices have a high-resolution color camera, capable of producing images at 30Hz. This camera is not used for motion tracking or localization, and we employ it in our system for adding color to the reconstructions.

Finally, both devices have a built-in depth sensor, based on structured infrared light. Both depth sensors on the two devices operate at a frequency of 5Hz, but produce data with different characteristics. The cell phone produces depth images similar in density and coverage to those from a Kinect (albeit at a lower, QVGA resolution). On the other hand, the tablet produces data which is much sparser, as well as with poorer coverage on IR-absorbing surfaces (see Fig. 4.3). The data on the tablet arrives directly in the form of a point cloud.
Point observations are calculated by using an IR camera and detecting features from an IR illumination pattern at sub-pixel accuracy. Converting to a depth image can be done by using the focal length of the tablet’s IR camera as a reference. However, this produces a depth image with significant gaps (Fig. 4.3b). Creating a depth image with fewer and larger pixels eliminates the gaps, at the cost of the loss of angular resolution for each returned 3D point. Any algorithm we use must be aware of the data peculiarity.

4.1.2 Integrated motion estimation

*Project Tango* devices come with a built-in 30Hz visual-inertial odometry (VIO) system, which we use as the main source of our pose estimation. The system is based on the work of Hesch et al. [38]. It fuses information from the IMU with observations of point features in the wide-angle camera images using a multi-state constraint Kalman filter [53]. The VIO system provides the pose of the device base frame $B$ with respect to a global frame $G$ at time $t$: $G^B_t$. We are also provided the (constant) poses of the depth camera $D$ and the color camera $C$ with respect to the device base, $b^D$ and $b^C$ respectively.

The first issue worth noting is that the VIO pose information, depth data, and color images all arrive asynchronously. To resolve this, when a depth or color data arrives with a timestamp $t'$, we buffer it and wait until we receive at least one VIO pose $G^B_t$ such that $t \geq t'$, and at least one pose such that $t \leq t'$. Then, we linearly interpolate for the pose between the two VIO readings.

The second issue is that like any open-loop odometry system, *Project Tango*’s VIO pose is subject to drift. This can result in model inaccuracies as the user revisits previously-scanned parts of the scene.
4.1.3 Depth measurement model

We have already presented our depth measurement model for an RGB-D camera in Section 3.1. Here, we use the same model, and present results for the depth cameras in Tango devices.

We found that the depth data returned by Tango devices exhibits a strong systematic bias, similar to the error described in Section 3.1.2, but in some cases, larger in magnitude. Using our depth calibration procedure, we can remove a significant amount of the bias in the depth reading. We collected a set training images, and example of which is shown in Fig. 4.4. Fig. 4.5 presents the results for the Project Tango mobile phone, which exhibited the worse systematic bias of the two devices. Each data point in the figure represents the RMS error of $z$ over all the pixels in a given checkerboard test image, versus the average depth of the checkerboard. The results with and without calibration are displayed. The figure shows that the RMS error is significantly lower when the polynomial compensation is performed on the depth images, and the error improvement becomes more pronounced as the distance between the object and the camera grows.

4.2 Theory

4.2.1 The signed distance function

We model the world based on a volumetric Signed Distance Function (SDF), denoted by $D : \mathbb{R}^3 \rightarrow \mathbb{R}$ [16]. For any point in the world $\mathbf{x}$, $D(\mathbf{x})$ is the distance to the nearest surface, signed positive if the point is outside of objects and negative otherwise. Thus, the zero isocontour ($D = 0$) encodes the surfaces of the scene. The Truncated Signed Distance
Figure 4.4: Two sets of training images used in the calibration for systematic bias in depth readings (top and bottom). Each training set consists of 3 images: the greyscale image (a) is used to compute a linear plane model (b) for the reference depth. The reference depth is compared to the measured depth data (c). We repeat for multiple checkerboard poses at different distances and view positions.

Function (TSDF), denoted by $\Phi : \mathbb{R}^3 \rightarrow \mathbb{R}$, is an approximation of the SDF which stores distances to the surface within a small truncation distance $\tau$ around the surface [16]:

$$
\Phi(x) = \begin{cases} 
D(x) & \text{if } |D(x)| \leq \tau \\
\tau & \text{if } D(x) > \tau \\
-\tau & \text{if } D(x) < \tau 
\end{cases} \quad (4.1)
$$

Consider an ideal depth sensor observing a point $p$ on the surface of an object, which is a distance $r$ away from the sensor origin.

$$
r \equiv \|p\| \quad (4.2)$$
Figure 4.5: RMS error in $z$ before and after depth bias calibration on the Tango cell phone. Each data point represents the average error over all checkerboard pixels in the $k$-th checkerboard image. Multiple checkerboard images at varying distances $z$ from the camera are used.

Let $\mathbf{x}$ be a point which lies along the observation ray. The points $\mathbf{p}$ and $\mathbf{x}$ are expressed with respect to the sensor’s frame of reference. We define the auxiliary parameters $d$ (distance from sensor to $\mathbf{x}$) and $u$ (distance from $\mathbf{x}$ to $\mathbf{p}$):

$$d \equiv \|\mathbf{x}\|$$

(4.3a)

$$u \equiv r - \|\mathbf{x}\|$$

(4.3b)

Thus, $u$ is positive when $\mathbf{x}$ lies between the camera and the observation, and negative oth-
Let \( \phi(x, p) \) be the observed TSDF for a point \( x \), given an observation \( p \). Near the surface of the object, where \(|u|\) is small (within \( \pm \tau \)), the signed distance function can be approximated by the distance along the ray to the endpoint of the ray. This is because we know that there is at least one point of the surface, namely the endpoint of the ray.

\[
\phi(x, p) \approx u \tag{4.4}
\]

Note that this approximation is better whenever the ray is approximately perpendicular to the surface, and worst whenever the ray is parallel to the surface. Because of this, some works \cite{10} instead approximate \( \phi \) by fitting a plane around \( p \), and using the distance to that plane as a local approximation:

\[
\phi(x, p) \approx -u(p \cdot n_p) \tag{4.5}
\]

where \( n_p \) is the local surface normal at \( p \). In general, the planar approximation (4.5) is much better than the point-wise approximation (4.4), especially when surfaces are nearly parallel to the sensor but computing surface normals is not always computationally feasible. Both approximations of \( \phi \) are defined only in the truncation region of \( d \in [r - \tau, r + \tau] \).

We are interested in estimating the value of the TSDF from subsequent depth observations at discrete time instances. Curless and Levoy \cite{16} show that by taking a weighted running average of the distance measurements over time, the resulting zero-isosurface of the TSDF minimizes the sum-squared distances to all the ray endpoints. Following their work, we introduce a weighting function \( W : \mathbb{R}^3 \to \mathbb{R}^+ \). Similarly to the TSDF, the weighting function is defined only in the truncation range. We draw the weighting function from the uncertainty
Figure 4.6: Observation ray geometry. The depth sensor is observing a point $p$ on the surface of an object. A point $x$, which lies on the observation ray, is a distance $d$ from the sensor origin, and a distance $u$ from the observation $p$.

The model of the depth sensor. The filter is defined by the following transition equations:

$$
\Phi_{k+1}(x) = \frac{\Phi_k(x)W_k(x) + \phi_k(x, p)w_k(x, p)}{W_k(x) + w_k(x, p)} \tag{4.6a}
$$

$$
W_{k+1}(x) = W_k(x) + w_k(x, p) \tag{4.6b}
$$

where for any point $x$ within the truncation region, $\Phi(x)$ and $W(x)$ are the state TSDF and weight, and $\phi$ and $w$ are the corresponding observed TSDF and observation weight according to the depth observation $p$. The filter is initialized with

$$
\Phi_0(x) = \text{undefined} \tag{4.7a}
$$

$$
W_0(x) = 0 \tag{4.7b}
$$

for all points $x$.

Ideally, the distance-weighting function $w(x, p)$ should be determined by the probability distribution of the sensor, and should represent the probability that $\phi(x, p) = 0$. It is possible
to directly compute the weight from the probability distribution of the sensor, and hence compute the actual expected signed distance function. In favor of better performance, linear, exponential, and constant approximations of $w$ can be used [10, 16, 57, 91].

In our work, we use a constant approximation. The function is only defined in the truncation region $d \in [r - \tau, r + \tau]$, and normalizes to 1:

$$\int_{r-\tau}^{r+\tau} w(x, p) \, dd = 1 \quad (4.8)$$

### 4.2.2 Dynamic truncation distance

Following the approach of Nguyen et al. [59], we use a dynamic truncation distance based on the noise model of the sensor rather than a fixed truncation distance. In this way, we have a truncation function:

$$\tau(z_{uv}) = \beta \sigma(r) \quad (4.9)$$

where $\sigma(r)$ is the standard deviation for an observation with a range $r$, and $\beta$ is a scaling parameter which represents the number of standard deviations of the noise we want to integrate over. For the sake of simplicity, we approximate the range-based deviation using the sensor model for the depth-depth based deviation $\sigma(z_{uv})$.

$$\sigma(r) \approx \sigma(z) \quad (4.10)$$

Using the dynamic truncation distance has the effect that further away from the sensor, where depth measurements are less certain and sparser, measurements are smoothed over a larger area of the distance field. Nearer to the sensor, where measurements are more certain and closer together, the truncation distance is smaller.
CHAPTER 4. LARGE-SCALE RECONSTRUCTION WITH A MOBILE DEVICE

The constant weighting function thus becomes

\[ w(x, p) = \frac{1}{2} \frac{1}{\tau(r)} = \frac{1}{2} \frac{1}{\beta \sigma(z)} \] (4.11)

where \( z \) is the \( z \)-component of \( p \).

4.2.3 Space carving

When depth data is highly noisy and sparse, the relative importance of negative data (that is, information about what parts of the space do not contain an obstacle) increases over positive data [20, 42]. This is because rays passing through empty space constrain possible values of the signed distance field to be positive at all points along the ray, whereas rays hitting objects only inform us about the presence of a surface very near the endpoint of the ray. In this way, rays carry additional information about the value of SDF.

So far, we have defined the TSDF observation function \( \phi \) only within the truncation region, defined by \( d \in [r - \tau, r + \tau] \). We found that it’s highly beneficial to augment the region in which the filter operates by \( [r_{\text{min}}, r - \tau) \), where \( r_{\text{min}} \) is the minimum observation range of the camera (typically around 40 centimeters). We refer to this new region as the space carving region. The entire operating region for filter updates thus becomes the union of the space carving and truncation regions: \( [r_{\text{min}}, r + \tau] \) (Fig. 4.7).

There are different ways in which to incorporate the space-carving observations into the filter. One way is to treat space-carving observations as an absolute constraint on the values...
Figure 4.7: The probability model for a given depth observation $p$. The model consist of a hit probability $P(d = r)$ and pass probability $P(d < r)$, plotted against the distance $d$ along the ray. We define two integration regions: a space carving region $[r_{\text{min}}, r - \tau)$ and the truncation region $[r - \tau, r + \tau]$, where $r_{\text{min}}$ is the minimum depth camera range, and $\tau$ is the truncation distance.

\[ \Phi_k+1(x) = \tau(r) \]  
\[ W_{k+1}(x) = 0 \]

This works reasonably well, but ends up being too aggressive - sometimes the sensor will produce false long-distance observations, which can incorrectly “clear” the accumulated TSDF values along the way.

Instead, we choose to incorporate space-carving observations into the TSDF filter by treating them as regular observations. To do so, we must assign a TSDF value $\phi$ and weight
\[ w(x, p) = \tau \quad \text{if } d \in [r_{\min}, r - \tau) \]  
\[ w(x, p) = w_{sc} \quad \text{if } d \in [r_{\min}, r - \tau) \]  

where \( w_{sc} \) is a fixed space-carving weight.

Space carving gives us two advantages: first, it dramatically improves the surface reconstruction in areas of very high noise, especially around the edges of objects (see Fig. 4.19). Further, it removes some inconsistencies caused by moving objects and localization errors. If space carving is not used, moving objects appear in the TSDF as permanent blurs, and localization errors result in multiple overlapping isosurfaces appearing. With space carving, old inconsistent data is removed over time.

### 4.2.4 Colorization

As in [10, 91], we create textured surface reconstructions by directly storing color as volumetric data. We augment our model of the scene to include a color function \( \Psi : \mathbb{R}^3 \to \text{RGB} \), with a corresponding weight \( G : \mathbb{R}^3 \to \mathbb{R}^* \). We assume that each depth observation ray also carries a color observation, taken from a corresponding RGB image. As in [10], we have chosen RGB color space for the sake of simplicity, at the expense of color consistency with changes in illumination.

Color is updated in exactly the same manner as the TSDF. The update equation for the
where for any point $x$ within the truncation region, $\Psi(x)$ and $G(x)$ are the state color and color weight, and $\psi$ and $g$ are the corresponding observed color and color observation weight.

In both [10] and [91], the color weighting function is proportional to the dot product of the ray’s direction and an estimate of the surface normal. But since surface normal estimation is computationally expensive, we instead use the same observation weight for both the TSDF and color filter updates:

$$g(x, p) = w(x, p) \quad (4.15)$$

We must also deal with the fact that color images and depth data are asynchronous. In our case, depth data is often delayed behind color data by as much as 30 milliseconds. So, for each depth scan, we project the endpoints of the rays onto the nearest color image in time to the depth image, accounting for the different camera poses due to movement of the device. Then, we use bilinear interpolation on the color image to determine the color of each ray.

### 4.2.5 Discretized data layout

We use a discrete approximation of the TSDF which divides the world into a uniform 3D grid of voxels. Each voxel contains an estimate of the TSDF and an associated weight. More precisely, we define the Normalized Truncated Signed Distance Function (NTSDF), denoted
by $\Phi : \mathbb{Z}^3 \rightarrow \mathbb{Z}$. For discrete voxel coordinate $v = [v_i, v_j, v_k]^T$, the NTSDF $\Phi(v)$ is the discretized approximation of the TSFD, scaled by a normalization constant $\lambda$. We similarly define a discretized weight, denoted by $\overline{W} : \mathbb{Z}^3 \rightarrow \mathbb{Z}^+$, which is the weight $W$ scaled by a constant $\gamma$.

\[ \Phi(v) = \lfloor \lambda \Phi(x) \rceil \quad (4.16a) \]
\[ \overline{W}(v) = \lfloor \gamma W(x) \rceil \quad (4.16b) \]
\[ v = \left\lfloor \frac{1}{s_v} x \right\rfloor \quad (4.16c) \]

where $s_v$ is the voxel size in meters, and $\lfloor \rceil$ is the round-to-nearest-integer operator.

In our implementation, the NTSDF and weight are packed into a single 32-bit structure. The first 16 bits are a signed integer distance value, and the last 16 bits are an unsigned integer weighting value. Color is similarly stored as a 32 bit integer, with 8 bits per color channel, and an 8 bit weight (Fig. 4.8). A similar method is used in [3, 12, 19] to store the TSDF.

We choose $\lambda$ in a way to maximize the resolution in the 16-bit range $[-2^{15}, 2^{15}]$. By definition, the maximum TSDF value occurs at the maximum truncation distance. The
truncation distance is a function of the reading range (4.9). The maximum reading range \( r_{\text{max}} \) is a fixed parameter of the depth camera usually chosen at around 3.5 to 4 meters. Thus, we can define \( \lambda \) as:

\[
\lambda = 2^{15} \tau (r_{\text{max}})
\] (4.17)

This formulation guarantees \( \Phi \) utilizes the entire \([-2^{15}, 2^{15}]\) range.

We similarly choose \( \gamma \) to maximize the weight resolution. By the definition of our sensor model, the lowest possible weight \( W_{\text{min}} \) occurs at the maximum truncation distance \( \tau (r_{\text{max}}) \). We define that weight to be 1, and scale the rest of the range accordingly:

\[
\gamma = \frac{1}{W_{\text{min}}}
\] (4.18)

This formulation guarantees \( W \geq 1 \).

### 4.2.6 Discretized data updates

We define the weighted running average filter (WRAF) as a discrete-time filter for an arbitrary state quantity \( X \) and weight \( W \) with observation \( x \) and observation weight \( w \) with the following transition equations:

\[
X_{k+1} = \frac{X_k W_k + x_k w_k}{W_k + w_k}, \quad X, x \in \mathbb{R}
\] (4.19a)

\[
W_{k+1} = W_k + w_k, \quad W, w \in \mathbb{R}^+
\] (4.19b)

In Section 4.2, we applied this filter to both the TSDF and color updates. However, since we store the data using integers, we need to define a discrete weighted running average filter (DWRAF) as a variation of the ideal WRAF where the state, observation, and weights are
The observation weight is a strictly positive integer, while the state weight is a non-negative-integer (allowing the filter to be initialized with zero weight, corresponding to an unknown state).

We further define the observation delta between the observation and the current state as:

$$\Delta x_k \equiv x_k - X_k$$

where $\Delta x \in \mathbb{R}$.

It can be shown that due to the rounding step in (4.20b), when the state weight is greater than or equal to twice the observation delta, then the state transition delta will be zero (the state remains the same).

$$w_k \geq 2|\Delta x_k| \implies X_{k+1} = X_k$$

This means that for any given state weight, there exists a minimum observation delta, and observations which are too close to the state will effectively be ignored. Since the state weight is monotonically increasing with each filter update, the minimum observation delta grows over time, causing the filter to ignore a wider range of observations.

Consider the example when the filter state $X \in [-2^{15}, 2^{15}]$ represents TSDF in the range $[-0.10, 0.10]$ meters. When the filter has received 100 observations, each with an observation
weight of 1, the minimum observation delta will be approximately 0.15 millimeters. At 10000 observations, the minimum observation delta is 1.5 centimeters, etc. If observations with higher weights go in, the minimum observation delta will grow even quicker.

The problem is much more significant with smaller discrete spaces, such as colors discretized in the $X \in [0,255]$ range. A filter with a state weight of 100 will require an observation delta of 50, or approximately 20% color difference, in order to change its state. Once the state weight reaches 510, the filter will enter a steady state, and no subsequent observations will ever perturb it.

One way to deal with this issue is to impose an artificial upper maximum on the state weight, and prevent it from growing beyond that with new updates. This guarantees that the minimum observation delta is bound. This is a straight-forward solution which might work for the distance filter, but not for the color filter, where the maximum weight would have to be set very low. We propose a different solution, which modifies the state transition equation 4.20b to guarantee a response even at high state weights:

$$X_{k+1} = \begin{cases} 
X_k + 1 & \tilde{X}_k - X_k \in (0,0.5) \\
X_k - 1 & \tilde{X}_k - X_k \in (-0.5,0) \\
\lfloor \tilde{X}_k \rfloor & \text{otherwise}
\end{cases}$$

(4.23)

This preserves the behavior of the DWRAF, except in the cases when the rounding would force the new state to be the same as the old one. In those cases, we enforce a state transition delta of 1 (or $-1$).

The three different filter behaviors (ideal WRAF, DWRAF, and modified DWRAF) are illustrated in Fig. 4.9. We simulate a filter update scenario where the filter starts with a state
Figure 4.9: Convergence behavior of the ideal weighted running average filter (WRAF), discrete WRAF, and modified discrete WRAF. The filter starts with a state weight of 5, and receives a series of updates with a constant observation weight $w$ of 1 and a constant observation $x$ of 100.

In our implementation, we use the modified DWRAF filter implementation for the NTSDF and color updates.
4.2.7 Dynamic spatially-hashed volume grid

Unfortunately, a flat volumetric representation of the world using voxels is incredibly memory intensive. The amount of memory storage required grows as $O(N^3)$, where $N$ is the number of voxels per side of the 3D voxel array. For example, at a resolution of 3cm, a 30m TSDF cube with color would occupy 8 Gigabytes of memory. Worse, most of that memory would be uselessly storing unseen free space. Further, if we plan on exploring larger and larger distances using the mobile sensor, the size of the TSDF array must grow if we do not plan on allocating enough memory for the entire space to be explored.

For a large-scale real-time surface reconstruction application, a less memory-intensive and more dynamic approach is needed. Confronted with this problem, some works have either used octrees [94, 96, 12], or use a moving volume [91]. Neither of these approaches is desirable for our application. Octrees, while maximally memory efficient, have significant drawbacks when it comes to accessing and iterating over the volumetric data [60]. Every time an octree is queried, a logarithmic $O(M)$ cost is incurred, where $M$ is the depth of the Octree. In contrast, queries in a flat array are $O(1)$. An octree stores pointers to child octants in each parent octant. The octants themselves may be dynamically allocated on the heap. Each traversal through the tree requires $O(M)$ heap pointer dereferences in the worst case. Even worse, adjacent voxels may not be adjacent in memory, resulting in very poor caching performance [14]. Like [60], we found that using an octree to store the TSDF data to reduce iteration performance by an order of magnitude when compared to a fixed grid.

Instead of using an Octree, moving volume, or a fixed grid, we use a hybrid data structure introduced by Nießner et al. [60]. The data structure makes use of two levels of resolution: volumes and voxels. Volumes are spatially-hashed [14] into a dynamic hash map. Each volume consists of a fixed grid of $N_v^3$ voxels, which are stored in a monolithic memory block.
Volumes are allocated dynamically from a growing pool of heap memory as data is added, and are indexed in a spatial 3D hash map [14] by their spatial coordinates. As in [14, 60] we use the hash function: \( \text{hash}(x, y, z) = p_1 x \oplus p_2 y \oplus p_3 z \mod n \), where \( x, y, z \) are the 3D integer coordinates of the chunk, \( p_1, p_2, p_3 \) are arbitrary large primes, \( \oplus \) is the xor operator, and \( n \) is the maximum size of the hash map.

Since volumes are a fixed size, querying data from the hybrid data structure involves rounding (or bit-shifting, if \( N_v \) is a power of two) a world coordinate to a chunk and voxel coordinate, and then doing a hash-map lookup followed by an array lookup. Hence, querying is \( \mathcal{O}(1) \) [60]. Further, since voxels within chunks are stored adjacent to one another in memory, cache performance is improved while iterating through them. By carefully pre-selecting the size of volumes so that they corresponding to \( \tau_{\text{max}} \), we only allocate volumetric data near the zero isosurface, and do not waste as much memory on empty or unknown voxels.

Additionally, each volume also contains a mesh segment, represented by a list of vertices and face index triplets. The mesh may also optionally contain per-vertex color. Each mesh segment corresponds to the isosurface of the NTSDF data stored in the volume. The layout of the data structure is shown in Fig. 4.10.

4.2.8 Gradient-based dense alignment

To account compensate for VIO drift, we calculate a correction to the VIO pose by aligning the depth data to the isosurface (\( \Phi = 0 \)) of the TSDF field. We denote this correction by \( \mathbf{T}_{\Phi} \), corresponding to the pose of the the global frame \( G \) with respect to the corrected global frame \( \overline{G} \), at time \( t \). This correction is initialized to identity, and is recalculated with each new depth scan which arrives, as described in the following subsection.
When a new point cloud \( P = \{ p_i \} \) is available, we calculate the predicted position of all the points, using the previous correction \( G_{G_t} T_{t-1} \) and the current VIO pose \( G_B T_t \):

\[
\overline{G} p_i = G_{G_t} T_{t-1} G_B T_t B D T_0 p_i
\]

Let there be some error function \( e(P) \) which describes the deviation of the point cloud from the model isosurface, and a corresponding transformation \( \Delta T \), which, when applied to
the point cloud, minimizes the error.

\[
\arg \min_{\Delta T} e(P) \tag{4.25}
\]

Once we obtain \(\Delta T\), we can apply it to the previous corrective transform in order to estimate the new correction:

\[
\overrightarrow{G}T_t = \Delta T \overrightarrow{G}T_{t-1} \tag{4.26}
\]

It remains to be shown how to formulate the error function and the minimization calculation. We do this by modifying the classic iterative closest point (ICP) problem \([13]\), using the gradient information from the voxel grid. For each point \(p_i\) in the point cloud \(P\), there exists some point \(m_i\), which is the closest point to \(p_i\) on the isosurface. We can therefore define the per-point error \(e(p_i)\) and total weighted error \(e(P)\) as:

\[
e(p_i) = \|m_i - \Delta T \overrightarrow{p}_i\|^2 \tag{4.27a}
\]

\[
e(P) = \sum_i (w_i e(p_i)) \tag{4.27b}
\]

Next, we must find an appropriate value for \(m_i\). The ICP algorithm accomplishes this by doing a nearest-neighbor search into a set of model points. Instead, we will use the TSDF gradient. By the TSDF definition, the distance from any point \(p_i\) to the closest surface is given by the TSDF value. The direction towards the closest point is directly opposite the TSDF gradient. Thus,

\[
m \approx - \frac{\nabla \Phi(p)}{\|\nabla \Phi(p)\|} \Phi(p) \tag{4.28}
\]

We can look up the gradient information in \(O(1)\) time per point. Therefore, this algorithm performs faster than the classical ICP formulation, which requires nearest-neighbor
computations.

We approximate the gradient $\nabla \Phi$ by taking the difference between the corresponding TSDF for that voxel and and its three positive neighbors along the $x$, $y$, and $z$ dimensions. Note that the approximations only holds when the TSDF has a valid value, which only occurs at distances up to the truncation distance $\tau$ away from the surface. This limits the convergence basin for the minimization to areas in space where $\Phi(p_i) < \tau$. However, as long as each new VIO pose has a linear drift magnitude less then $\tau$ from the last calculated aligned pose, we are generally able to converge to the right corrected position. In practice, the approximation in (4.28) becomes better the closer we are to the surface. Thus, we perform the minimization iteratively, recalculating the corresponding points at each iteration.

Since we are aligning against a persistent global model, this effectively turns the open-loop VIO pose estimation into a closed-loop pose estimation. We found that in practice, the VIO pose is sufficiently accurate for reconstructing small (room-sized) scenes, if areas are not revisited. Using the proposed dense alignment method, we are able to correct for drift and “close” loops in mid-sized environments such as an apartment with several rooms. Fig. 4.11 shows a comparison between the visual-inertial odometry (VIO) trajectory, as well as the corrected trajectory calculated using the dense alignment (DA). Fig. 4.12 shows different views of the scene from the same experiment. Enabling dense alignment results in superior reconstruction quality.

### 4.3 Scan integration

Updating the data structure requires associating voxels with the observations that affect them. We discuss two algorithms which can be used to accomplish this: *voxel traversal* and
4.3.1 Voxel traversal

The voxel traversal algorithm is based on raytracing [3], and is described by Alg. 2. For each depth measurement, we begin by performing a raytracing step on the coarse volume grid to determine the affected volumes and make sure all of them are allocated. Next, we raytrace along the voxel grid of each affected volume, and update each visited voxel’s NTSDF (if they lie in the voxel carving region), or NTSDF and color (if they lie in the truncation region). We can choose the raytracing range for each observation to either \([r_{\text{min}}, r + \tau]\) or \([r - \tau, r + \tau]\),

Figure 4.11: Trajectories calculated online using Project Tango’s visual-inertial odometry (VIO), as well as the proposed dense alignment correction (VIO + DA).
Figure 4.12: Comparison of reconstructions using the trajectories in Fig. 4.11. Using dense alignment results in superior reconstruction quality.

depending on whether we want to perform voxel carving or not. The latter range will only allocate volumes and traverse voxels inside the truncation region, gaining speed and memory at the cost of reconstruction quality. For each voxel visited by the traversal algorithm, we
Algorithm 2 Voxel traversal integration

1: ▷ For each point observation:
2: for p ∈ P do
3:    ▷ Find all volumes intersected by observation.
4:       V ← RaytraceVolumes()
5:    ▷ Ensure all volumes are allocated
6:       AllocateVolumes(V)
7:    ▷ Find all voxels intersected by observation.
8:       X ← RaytraceVoxels()
9:    ▷ For each intersected voxel:
10:       for v ∈ X do
11:           d_in ← ∥v_in - o∥ ▷ Voxel entry range
12:           d_out ← ∥v_out - o∥ ▷ Voxel exit range
13:           d ← 0.5(d_out - d_in) ▷ Effective voxel range
14:           r ← ∥p∥ ▷ Measured range
15:           τ ← τ(r) ▷ Truncation distance
16:           if d ∈ [τ_min, r - τ) then
17:              ▷ Inside space carving region:
18:              UpdateNTSDF(τ, w_SC)
19:           else if d ∈ [r - τ, r + τ] then
20:              ▷ Inside truncation region:
21:              w ← 1/3τ ▷ Observation weight
22:              c ← Color{u, v}) ▷ Observation color
23:              UpdateNTSDF(u, w)
24:              UpdateColor(c, w)

calculate the effective voxel range \(d\), which is average of the entry range \(d_{in}\) and exit range \(d_{out}\), corresponding to the distances at which the ray enters and exits the voxel.

Let \(\rho_v\) be the linear voxel density, equal to the inverse of the voxel size \(s_v\). the computational complexity of the voxel traversal algorithm becomes \(O(|P|\rho_v)\), where \(|P|\) is the number of observations in the point cloud \(P\) (or, if using a depth image, the number of pixels with a valid depth reading).

This formulation of voxel traversal makes the approximation that each depth observation can be treated as a ray. As a result, as the depth rays diverge further away from the sensor,
some voxels might not be visited. A more correct approximation would be to treat each depth observation as a narrow pyramid frustum, and calculate the voxels which the frustum intersects. However, this comes with a significantly greater computational cost.

4.3.2 Voxel projection

The voxel projection algorithm, described by Alg. 3, works by projecting points from the voxel grid onto a 2D image. As such, it requires that the depth data is in the form of a depth image, with a corresponding projection matrix. Voxel projection has been used by [10, 42, 59] for 3D reconstruction. The algorithm is analogous to shadow mapping [76] from computer graphics, where raycast shadows are approximated by projecting onto a depth map from the perspective of the light source, except in our case, shadows are regions occluded from the depth sensor.

In order to perform voxel projection, we must iterate over all the voxels in the scene. This is, of course, both computationally expensive and inefficient. The majority of the iterated voxels will not project on the current depth image. Moreover, due to the nature of the two-tier data structure, some voxels of interest might not have been allocated at all. Thus, as a preliminary step, we determine the set of all potentially affected volumes $V$ in the current view. We do this by calculating the intersection between the current depth camera frustum and the volume grid. The far plane of the frustum is at a distance of $r_{\text{max}} + \tau(r_{\text{max}})$, where $r_{\text{max}}$ is the maximum camera range.

Next, we iterate over the voxels of all potentially affected volumes. We approximate the effective voxel range $d$ as the distance from the voxel’s centroid to the camera origin. By projecting the voxel onto the depth image, we can obtain a pixel coordinate and corresponding depth $z_{uv}$, from which we can calculate the observed range $r$. The rest of the algorithm
Algorithm 3 Voxel projection integration

1: \(\triangleright\) Find all volumes in current frustum.
2: \(V \leftarrow \text{FrustumIntersection}()\)
3: \(\triangleright\) Ensure all volumes are allocated.
4: \(\text{AllocateVolumes}(V)\)
5: \(\triangleright\) For each intersected volume:
6: for \(V \in V\) do
7: \(\triangleright\) For each voxel centroid in volume:
8: for \(v_c \in V\) do
9: \(\triangleright\) Effective voxel range
10: \(d \leftarrow \|v_c - o\|\)
11: \(\{u, v\} \leftarrow \text{Project}(v_c)\)
12: \(r \leftarrow \|p\|\)
13: \(\tau \leftarrow \tau(r)\)
14: if \(d \in [\tau_{\text{min}}, r - \tau]\) then
15: \(\triangleright\) Inside space carving region:
16: \(\text{UpdateNTSDF}(\tau, w_{\text{SC}})\)
17: else if \(d \in [r - \tau, r + \tau]\) then
18: \(\triangleright\) Inside truncation region:
19: \(w \leftarrow \frac{1}{2\tau}\)
20: \(c \leftarrow \text{Color}(\{u, v\})\)
21: \(\text{UpdateNTSDF}(u, w)\)
22: \(\text{UpdateColor}(c, w)\)
23: \(\triangleright\) Clean up volumes that did not receive updates.
24: \(\text{GarbageCollection}(V)\)

proceeds analogous to voxel traversal: we check the region that this voxel belongs to, and conditionally update its NTSDF and color.

Finally, we perform a garbage collection step, which iterates over all the potentially affected volumes \(V\) and discards volumes that did not receive any updates to any of their voxels.

The computational complexity of the algorithm is \(O(|V| \rho_v^3)\), where \(|V|\) is the number of volume candidates, and \(\rho_v^3\) is the number of voxels in a unit of 3D space.

This formulation of voxel projection approximates each voxel as its centroid point. Thus,
Table 4.1: Comparison of integration algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Voxel traversal</th>
<th>Voxel projection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity:</td>
<td>$O(</td>
<td>P</td>
</tr>
<tr>
<td>Approximations:</td>
<td>Depth observations approximated as rays</td>
<td>Voxel range approximated by the voxel centroid; Each voxel projects to a single depth observation.</td>
</tr>
<tr>
<td>Approximation effects:</td>
<td>Integration gaps far from the camera</td>
<td>Aliasing effects with larger voxels</td>
</tr>
</tbody>
</table>

the estimation of the effective voxel range can have an error of up to $\sqrt{3}/2 \ s_v$ (half the voxel’s diagonal). Furthermore, each voxel will only project onto a single depth pixel, receiving a single update. Compare that to voxel traversal, where a single voxel might be updated by multiple depth rays passing through it, thus averaging the observations from all of them. At the resolution that we were interested in (around 3cm) these approximations lead to worse reconstruction quality than the approximations used in voxel reversal integration.

We summarize the properties of the two algorithms in Table 4.1.

### 4.3.3 Parallelization

We present a method to parallelize the scan integration algorithms, so that scan integration can take advantage of multi-threading and the multi-core architecture of the mobile devices.

Voxel projection integration is straightforward to implement in a parallel way. The frustum intersection and volume allocation are carried out serially. The outer loop in Alg. 3 is parallelized so that each volume receives its own job. Volumes are independent of one another - thus, each job owns its own data, and no synchronization or locking is required. The only data shared between the jobs is the depth image that the volumes project onto and the camera extrinsic and intrinsic parameters. All of these are constant during any given
Voxel traversal integration is more challenging to parallelize. We describe the parallelization problem in terms of a graph problem. Consider the bipartite graph in Fig. 4.13, split between the observation set $P$ and the volume set $V$. The serial version of the voxel traversal algorithm we have presented in Alg. 2 iterates over all the points in $P$ and finds their corresponding affected volumes from $V$. Thus, we can think of the algorithm as building a directed bipartite graph, where the direction is from observation to volume.

A naive way to parallelize the traversal problem is to follow with this direction of the bipartite graph and instantiate a single job for each observation. This has the drawback that each job needs write access to multiple volumes, and thus, a locking mechanism is required in order to implement concurrency. Instead, we reorganize the problem by inverting the direction of the bipartite graph so that each volume points to all the observations which affect it. We perform this re-indexing step serially. Next, we instantiate a job for each affected volume, and integrate the data from all the observations which it points to. Thus, we have restructured the problem so that, similarly to parallelized voxel projection, we have the volume as the unit of job separation. Choosing volumes over observations as the job unit has another advantage: in the datasets we used, there were anywhere between one and two orders of magnitude more observations than corresponding affected volumes. Having fewer jobs that run for longer is preferable over having a greater number of shorter jobs, as there is an overhead incurred by creating a job thread, as well as frequent context switching.

It remains to be shown how the graph re-indexing step affects the computational complexity of the voxel traversal algorithm, which we have so far established to be $O(|P| \rho_v)$. The added complexity of this step is equal to the number of edges in the graph. Since calculating an edge requires a raytracing operation over the coarse volume grid for a given observation,
this gives us a total complexity of $O(|P| \rho_{vol})$, where $\rho_{vol}$ is the linear volume density. By
definition, each volume is bigger than the voxels it contains, and therefore $\rho_{vol} < \rho_v$. Thus,
the total computational complexity remains the same. In practice, raytracing over the coarse
grid is much faster than raytracing over the voxel grids; therefore, the computational over-
head added by the re-indexing step is minimal, and outweighed by the gain in parallelizing
the integration.

4.4 Mesh extraction and rendering

4.4.1 Online mesh extraction

We generate a mesh of the isosurface using the Marching Cubes algorithm [45]. The mesh
vertex location inside the voxel is determined by linearly interpolating the TSDF of its neigh-
bors. Similarly, as in [10, 91], colors are computed for each vertex by trilinear interpolation
of the color field. We can optionally also extract the per-face normals, which are utilized in
rendering.
We maintain a separate mesh for each segment for each instantiated volume (Fig. 4.10). Meshing a given volume requires iterating over all its voxels, as well as the border voxels of its \(7\) positive neighbors (volumes with at least one greater index along the \(x\), \(y\), or \(z\) dimension, and no smaller indices). The faces which span the border across to positive neighbors are stored with the mesh of the center volume. Therefore, the top-most, right-most, and forward-most bounding vertices of each mesh segment are duplicates of the starting vertices of the meshes stored in its positive neighbors. This is done to ensure that there are no gaps between the mesh segments, and that when rendered, the mesh appears seamless. Vertex duplication isn’t strictly necessary, as we could accomplish this by indexing directly into the neighboring mesh. However, direct indexing would introduce co-dependencies between parallel jobs.

We perform mesh extraction every time a new depth scan is integrated. As we have shown in Alg. 2 and Alg. 3, the voxel integrations algorithms estimate a set of volumes \(\mathcal{V}\), potentially affected by the depth data. Mesh extraction is performed only for those volumes. Thus, the computational complexity of the meshing algorithm becomes \(\mathcal{O}(|\mathcal{V}| 3^v)\). We parallelize meshing similarly to how we parallelize integration, by instantiating a single marching cubes job per volume, and letting jobs run concurrently. Since each job owns the data that it is modifying (the triangle mesh), there are no concurrency issues.

Parallelized meshing of only the affected volumes allows us to maintain an up-to-date representation of the entire isosurface in real time. Still, we observed that a lot of time is spent re-meshing volumes which are completely empty. These volumes occur mostly between the sensor and the surface, especially when we have voxel carving available. To further decrease the time spent for meshing, we propose a lazy extraction scheme which prunes a large portion of the volumes. To do so, we augment the integration algorithms to keep track of a per-volume flag signifying whether the isosurface has jumped across voxels.
since the last scan integration. The flag is set to true when at least one voxel in the current volume receives an NTDF update which changes its NTDF sign. The flag is cleared after each re-meshing.

The effect of the lazy extraction flag is that volumes with no iso-surface, or where the isosurface did not move during the last scan integration, will not be considered for meshing. This decreases the time spent extracting meshes, at the cost that we might not have the most up-to-date mesh available. As we mentioned earlier, the position of the vertex inside a voxel is calculated using a linear interpolation with the NTDF of its neighbors. Thus, even if the isosurface doesn’t jump between voxels, a depth scan integration might move the vertices within each voxel. Enabling lazy extraction means that at worst, each vertex can be up to $\frac{\sqrt{3}}{2} s_v$ (half a voxel diagonal) away from its optimal position. Further, lazy extraction does not consider updates to color, so color changes that do not come with occupancy changes will not trigger re-meshing.

In practice, we found that the sub-optimality conditions described above are triggered very rarely, and that lazily-extracted meshes were indistinguishable from the optimal meshes during real-time experiments.

At the end of each dataset collection trial, the user may want to save a copy of the entire mesh. In those cases, we run a single marching cubes pass across all the volumes in the hash map to produce a single, non-segmented mesh. Thus the final output mesh does not incur any of the approximations we made for real-time meshing (duplicate vertices and lazy extraction). After the final mesh is extracted, we perform (optional) mesh simplification via the quadric error vertex collapse algorithm [27], which reduces the number of mesh faces by a user-controllable amount. Finally, we remove any isolated vertices which have few neighbors within some small threshold radius. These vertices typically occur when the sensor has a
false depth readings, occasionally resulting in “speckle” artifacts floating around the scene.

4.4.2 Rendering

We render the mesh reconstruction using only simple fixed-function pipeline calls on the devices graphics hardware. To speed up rendering, we only consider mesh segments which are inside the current viewport’s frustum. This is done using frustum culling, analogous to how we use it in the voxel projection algorithm (Alg. 3). Further, we utilize a simple level-of-detail (LOD) rendering scheme. Mesh segments which are sufficiently far away from the camera are rendered as a single box, whose size and color are specified by the mesh segment’s bounding box and average vertex color, respectively. “Sufficiently far away” is determined by checking whether the projection of the corresponding mesh volume exceeds a certain number of pixels on the screen.

4.5 Offline processing

4.5.1 Offline pose estimation

In larger scenes where the odometry can drift more, the approach described in Section 4.2.8 will not be able to account for drift errors. In situations like this, we perform an offline post-processing step on device. The post processing requires that we record the inertial, visual, and depth data to disk. Given that the Project Tango devices have sufficient persistent memory available (60GB and 120GB for the cell phone and tablet, respectively), this is not an issue. Once the data collection is finalized, we replay the data and perform visual-inertial bundle-adjustment on the trajectory [55]. The bundle adjustment solves a nonlinear weighted least-squares minimization over the visual-inertial data to jointly esti-
mate the device pose and the 3D position of the visual features. Correspondences between non-consecutive keyframes (loop-closure) are established through visual feature matching [46]. Once the bundle-adjusted trajectory is calculated, we rebuild the entire TSDF map using the recorded depth data, and extract the final surface once.
We present the results of this process in Fig. 4.14. We recorded a single, large dataset (approximately 200 meters trajectory length). Fig. 4.14a shows a top-down view of the reconstruction performed online. Fig. 4.14b shows the reconstruction after performing the offline bundle adjustment. We overlaid it on the building schematic for reference. Finally, Fig. 4.14c shows a side view of the offline reconstruction. The surface area of the entire reconstructed mesh is approximately 740 square meters.

4.5.2 Multi-dataset reconstruction

Given an accurate trajectory and bias-free measurement model, we found that the biggest limiting factor for the quality of the reconstruction is the number of depth images taken. Each new depth image either adds information about an unobserved space, fills in small gaps in the existing reconstruction, or refines the estimation of the reconstruction’s isosurface. As we noted before, depth cameras such as the Kinect provide data with a rate and density much higher than the Project Tango mobile devices. We estimated that the Yellowstone tablet, for example, might receive two orders of magnitudes less depth observations for a comparable dataset collection trial. On the other hand, recording very long datasets is limited by battery life and is cumbersome for the operator, who has to walk slowly and spend a long time scanning.

The reconstructions shown so far, both in our work and in the works we have referenced, are created using a single dataset collection trial. We extend our workflow by allowing for automated fusing multiple datasets into a single reconstruction. Each dataset can be recorded from a different starting location and cover different areas of the scene, as long as their is sufficient visual overlap so that they can be co-registered together. We repeat the offline procedure described in Section 4.5.1 for each dataset. Then, we query keyframes from each
dataset against keyframes from the other ones, until we have sufficient information to calculate the offset between each individual trajectory, bringing them into a single global frame. Finally, we insert the depth data from each dataset into a unified TSDF reconstruction.

The results of this process are shown in Fig. 4.15 and Fig. 4.16. We asked two different operators to create two full reconstructions of an apartment scene. This resulted in four individual reconstructions, each with varying degrees of missing data. Fig. 4.15 shows each individual reconstruction (labeled “A” through “D”), as well as the final combined reconstruction. The final mesh has significantly fewer gaps. Fig. 4.16 shows the incremental resulting mesh at the different stages of the reconstruction, as each new dataset is fused into the previous stage. Each new stage fills in more mesh holes and refines the estimate of the
4.6 Experiments

4.6.1 Qualitative experiments

We tested the qualitative performance of the system on a variety of data sets. So far, we’ve shown reconstructions from the mobile phone data in Fig. 4.1a. In this section, we further present a reconstruction in Fig. 4.17, which demonstrates the effectiveness of our depth compensation model.

We have also shown several reconstructions from tablet data: the “Apartment” scene (Fig. 4.1b, Fig. 4.1c, Fig. 4.12, Fig. 4.15, Fig. 4.16) and “Corridor scene” (Fig. 4.14). In this section, we further present results from an outdoor reconstruction with tablet data (Fig. 4.18). Since the depth sensor is based on infrared light, outdoor reconstructions work better in scenes without direct sunlight.

We also investigated the effects of the different scan integration algorithms on the reconstruction quality using tablet data. We examined four combinations in total: voxel traversal vs. voxel projection, with voxel carving enabled or disabled (Fig. 4.19). We found that voxel traversal gives overall better quality at the resolution we are working with. Enabling voxel carving (for both traversal and projection algorithms) removes some random artifacts, and significantly improves reconstruction quality around the edges of objects).

Finally, we wanted to verify the applicability of the presented method with a third, non-Project Tango data source. We chose a the “Freiburg” datasets, recorded with a high-quality RGB-D camera and ground-truth trajectories from a motion-capture system [83]. The reconstruction results are presented in Fig. 4.20.
Figure 4.17: Comparison of model built with raw cell phone depth data (left) vs. compensated depth data (right). Using compensated data results in cleaner walls and object detail.
Figure 4.18: Outdoors reconstruction using tablet data, performed during an overcast day. Top-down view (b) also shows the device trajectory, in white.
Figure 4.19: Comparison of reconstruction quality using the different integration algorithms (voxel traversal vs. voxel projection) with and without voxel carving applied. The reconstructions was performed with tablet data.
4.6.2 Quantitative experiments

We performed several qualitative experiments to analyze the performance of the different algorithms we use. In the first experiment, we compare the number of grid volumes which are affected by each depth scan. As we discussed in Section 4.3, this number is important because we would like to keep our computations focused on a small number of local volumes. The results are shown in Fig. 4.21. Voxel projection uses frustum culling to select
Figure 4.21: Top: analysis of the number of affected volumes per depth scan. We compare the voxel traversal vs. voxel projection algorithms. The results for voxel traversal depend on whether voxel carving has been enabled. Bottom: comparison of the number of meshed volumes vs. all the affected volumes. The difference between the two is due to the lazy meshing scheme.

volume candidates, which is a conservative approximation - thus, it ends up considering more candidates overall than voxel traversal. When using voxel traversal, the number of volume candidates depends on whether voxel carving is enabled. Regardless of the choice of algorithm, integration is performed on each candidate volume. However, as we discussed before, meshing is performed lazily, and might prune some candidate volumes. We demonstrate this in Fig. 4.21 (bottom). We overlaid the number of meshed volumes over the number of total integrated volumes over time. As can be seen, voxel projection benefits the most from the lazy meshing scheme.
We also analyzed the exact run-time of voxel projection and voxel traversal, with and without carving enabled. We carried out the experiments with two sets of data (cell phone dataset and tablet dataset), since depth data affects algorithm performance. We further carried out the experiment on both a mobile device and on a desktop machine, to determine the effect of the mobile constraint. The results are shown in Table 4.2 (tablet dataset) and Table 4.3 (cell phone dataset). The tables show the average and standard deviation of the total processing time for each depth scan, in milliseconds, including: depth scan compensation, dense alignment, scan integration, and mesh extraction. The integration and extraction steps are performed using the parallelized algorithms described in Section 4.3.3. We used a voxel size of 3cm, and $16 \times 16 \times 16$ voxels per volume.

The timing experiments reveal that when tablet data is used, all of our insertion algorithms perform in real time at 5Hz or higher. Thus, we can choose the algorithm which gives

### Table 4.2: Scan integration time [ms] (tablet dataset)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mobile</th>
<th>Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voxel projection</td>
<td>Carving</td>
<td>83.7 ± 13.0</td>
</tr>
<tr>
<td></td>
<td>No carving</td>
<td>72.1 ± 10.8</td>
</tr>
<tr>
<td>Voxel traversal</td>
<td>Carving</td>
<td>121.9 ± 21.1</td>
</tr>
<tr>
<td></td>
<td>No carving</td>
<td>85.6 ± 12.6</td>
</tr>
</tbody>
</table>

### Table 4.3: Scan integration time [ms] (cell phone dataset)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mobile</th>
<th>Desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voxel projection</td>
<td>Carving</td>
<td>80.2 ± 8.0</td>
</tr>
<tr>
<td></td>
<td>No carving</td>
<td>67.5 ± 6.9</td>
</tr>
<tr>
<td>Voxel traversal</td>
<td>Carving</td>
<td>211.1 ± 67.9</td>
</tr>
<tr>
<td></td>
<td>No carving</td>
<td>113.4 ± 31.6</td>
</tr>
</tbody>
</table>
Figure 4.22: Comparison of memory consumption between our spatially-hashed volume data structure (SH) vs a fixed-resolution grid approach (FG). We compare two datasets: from a Tango cell phone device (a) and from a Kinect camera (b). In the case of the Kinect camera, we tested both a short-range insertion (observations up to 2m) and a long-range one (observations up to 5m). Note the logarithmic scale on the right graph. A timestep unit is equal to the time between receiving two depth images in the system.

us the best quality (voxel traversal with carving). When using a the cell phone dataset, voxel projection takes around the same time, but voxel traversal is significantly more expensive. This is because the number of observations in a the cell phone depth scan is much higher (Fig. 4.3). At this resolution, we cannot run the best-quality algorithm, so we choose either projection with carving or traversal without carving, depending the type of quality approximation we want to achieve.

Finally, we profiled the memory performance of our system on two datasets, from a Project Tango device (Fig. 4.22a) and a “Freiburg” RGB-D dataset [83] (Fig. 4.22b). For the Freiburg dataset, we tested two different maximum camera ranges (2 meters and 5
meters). The figures show the total memory consumption using the spatially hashed volume grid data structure described in Section 4.2.7. We used a voxel size of 3cm, and $16 \times 16 \times 16$ voxels per volume. For reference, we also include the memory for a static, 3cm fixed-grid approach.
Chapter 5

Conclusion

In Chapter 2, we described a modular, open-source indoor navigation system for quadro-tors MAVs. The system works in indoor environments, where the obstacles are mostly rectilinear and the floors are piecewise linear. The system is self-contained and can work onboard a MAV. The minimum sensor requirements are a laser scanner and an IMU.

We described the elements of the system, which range from low-level to high-level computations and behaviors: state estimation, control, path-planning, teleoperation, localization and mapping.

We demonstrated that the navigation estimation system can be used to build 2D and 3D maps, comparable in accuracy to maps built using ground vehicles with odometric sensors. We further demonstrated that we can provide easy and intuitive interactions between the MAV and the human operator for teleoperation and waypoint control.

The major focus of our work and contribution is a well documented, open-source release of all our software and hardware. Researchers can download the test data and recreate our experiments.
In Chapter 3, we presented a system for scene reconstruction with an RGB-D camera, and its application to an MAV quadrotor.

We began by detailing a calibration procedure and uncertainty model for depth readings of RGB-D cameras. The methods we described allow 3D points constructed from depth images to be treated as zero-mean multi-variate Gaussian distributions with a known covariance matrix. This is of interest to any system which performs calculations on RGB-D data where the precision and accuracy are important. We demonstrate experimental evidence of how the calibration procedure affects visual odometry and mapping applications, and demonstrate the predictive power of our uncertainty estimation model, which is able to estimate uncertainties around object edges better than the previously published formulations in this field. The calibration procedure requires that a checkerboard is placed in multiple distances and locations from the camera to obtain dense data, and is reasonable for distances up to 4-5 meters.

Further, we presented a visual odometry system for RGB-D cameras. The system uses sparse features which are registered against a persistent model of bounded size. The model is updated through a probabilistic Kalman filter framework. In order to achieve this, we developed a formulation for the 3D uncertainty in sparse features in RGB-D images, based on a Gaussian mixture model of readings in a local image window.

Finally, we presented a place recognition procedure which is used to find correspondences and relative transformations between non-consecutive keyframes for doing pose-graph SLAM. We present two approaches - an exhaustive one and a heuristic one, and evaluate the recognition rate of the latter.

An implementation of our system, developed for use with the ROS [69] framework, is available for download under a free, open-source license from our website (http://robotics.
In Chapter 4 we demonstrated a fully-integrated system capable of creating and rendering large-scale 3D reconstructions of scenes in real-time using only the limited computational resources of a Project Tango device. We accomplish this without any GPU computing, and use only the fixed function pipeline to render the scene. We do this out of necessity to meet the computing requirements of a mobile device. We discuss our contributions in terms of sensor calibration, discrete filtering, trajectory correction, data structure layout, and parallelized TSDF integration and meshing algorithms. Our work is heavily influenced by the characteristics of the depth data we receive (high noise and bias, lower frequency, variable observation density) We have successfully applied it to two different mobile devices, as well as datasets from other RGB-D sensors, to efficiently produce accurate geometric models.

Where appropriate, we augment our system with post-processing tools to overcome limitations in the sensors and the device. Notably, we employ place recognition and sparse feature-based bundle adjustment to correct very large trajectories, as well as merge multiple datasets, in order to get the best possible model and coverage. This can be seen as one of the major potential areas of improvement. Further research into this area should allow for the depth measurements, and dense alignment information they provide, to be closely integrated with the sparse features for a more robust SLAM solution. This can include methods for real-time mesh deformation for long trajectories, eliminating the need for post-processing.
5.1 Future work

A current limitation of the scene reconstruction solutions we have described is that we do not evaluate the accuracy of the reconstructions in a measurable way. Certain characteristics of the system are easier to measure - algorithm runtime, data structure memory efficiency, or accuracy of the pose tracking. The reconstructed scenes, however, have not been compared to ground truth data in a meaningful way. Part of the challenge here is that accurate ground truth data for indoor scenes is hard to obtain.

Another area we are interested in pursuing is large-scale, life-long reconstruction using multiple agents. While Section 4.5.2 described some progress towards multi-dataset reconstructions, there remain a lot of challenges in combining data from multiple agents in a large environment. A major issue is dealing with non-static objects in the scene.

As more and more consumer and research devices come with embedded depth sensors, we expect our work to be of interest to developers and researchers working on human or robotic navigation, augmented reality, and gaming.
Bibliography


[61] Andreas Nüchter, Kai Lingemann, Joachim Hertzberg, and Hartmut Surmann. 6D
2007.

[62] Søren Maagaard Olesen, Simon Lyder, Dirk Kraft, Norbert Krüger, and Jeppe Barsøe
Jessen. Real-time extraction of surface patches with associated uncertainties by means

[63] Jae-Han Park, Yong-Deuk Shin, Ji-Hun Bae, and Moon-Hong Baeg. Spatial Uncertainty

[64] Kaustubh Pathak, Andreas Birk, Jann Poppinga, and Sören Schwertfeger. 3d forward
sensor modeling and application to occupancy grid based sensor fusion. In *Intelligent

[65] Pierre Payeur, Patrick Hébert, Denis Laurendeau, and Clément M Gosselin. Prob-
abilistic octree modeling of a 3d dynamic environment. In *Robotics and Automation,

[66] Mihail Pivtoraiko and Alonzo Kelly. Generating near minimal spanning control sets
for constrained motion planning in discrete state spaces. In *Intelligent Robots and


