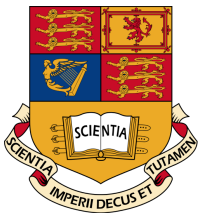


Imperial College London

Department of Computing

Semi-Dense and Dense 3D Scene Understanding in Embedded Multicore Processors

Author:
Andrew Jack

Supervisor:
Prof. Paul Kelly

Co-Supervisor:
Dr. Luigi Nardi

Second Marker:
Dr. David Ham

June 2015

Abstract

Getting lost is never an option. This especially made difficult with little or no prior knowledge of the environment, but this is the very challenge faced by robots and augmented reality systems. This problem of 'Simultaneous Location and Mapping' is still an unsolved problem, with a variety of approaches. Popular methods include dense scene reconstruction or semi-dense methods utilising some subset of the input.

We compare and contrast these two methods, as well as investigate the pipeline structure (sequential and parallel). We realise this comparison in KinectFusion (specifically the KFusion implementation) and LSD-SLAM, within the SLAMBench framework, which enables a *holistic* comparison of three metrics: trajectory error (ATE), FPS and energy per frame. We investigate on a desktop platform (x86) and an embedded platform (ARM Cortex-A15/A7, 'ODROID XU3').

We integrate LSD-SLAM into SLAMBench, showing it *can* perform substantially better in all metrics. However, we also show KFusion is reasonably robust in all three metrics. We perform some design space exploration, which highlights a variety of trade-offs between ATE and FPS, by controlling sparsity parameters and frequency at which the scene is summarised through the use of key-frames.

Furthermore, emphasise the high dependency of the results on the datasets used, particularly those with minimal lighting and textures. Finally, we highlight the benefit of a parallel architecture in achieving real-time tracking.

Acknowledgements

I would like to thank the following people:

- My supervisor Prof. Paul Kelly for his support and wisdom throughout this project.
- Dr. Luigi Nardi for his constant assistance and guidance throughout.
- Emanuele Vespa, and Zeeshan Zia for the countless number of helpful discussions we have had throughout this project.
- Dr. David Birch for his helpful proof-reading comments and suggestions.
- My house-mates, friends and family for their friendship and support.
- My late Father, and Mother for their endless love, care and encouragement.

"My flesh and my heart may fail,
but God is the strength of my heart
and my portion forever."
Psalm 73v26

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Motivation and Objectives

Computers are increasingly infiltrating our world as machines, robots, are beginning to be able to move and interact with us, as well as performing complex tasks.

The domain of robotics is not limited to physical machines moving within our world. Robotic principles apply to augmented reality[1] as well.

There are many challenges for a robot to fully interact with us. Some primary issues are [2] [3]:

- Sensors and Actuators - so they can interface with our world.
- Battery Technology - for the ability to be portable.
- Software and Algorithmic - providing their 'brain'.

The first two challenges, are mechanical and chemical issues respectively. We will, however, focus on the software and algorithmic challenges facing robotisits' today.

Robotic software, the nerve centre of the robot, has to manage how the robotic interacts with the world. A critical aspect is knowing its current location and map of the environment. Generally, neither are known when a robot first 'wakes up', hence the location and map need to be determined, *concurrently*. In the robotics domain this is known as: **S**imultaneous **L**ocation and **M**apping, *SLAM*.

SLAM is considered the prerequisite of autonomous robotics [4], and in its general form it is unsolved [5]. Therefore it is an important aspect of robotics to investigate and solve. As such, SLAM is an active research area, with new algorithms and ideas being frequently published. These newer additions to the field are only being compared to a select subset of the state-of-the-art algorithms, with comparisons primarily with reference to performance and accuracy.

An often neglected aspect of analysis is the energy consumption. This is very important for applications with a limited power supply such as a battery. By comparing

[1] the use of technology which allows the perception of the physical world to be enhanced or modified by computer-generated stimuli perceived with the aid of special equipment [1]

algorithms with respect to three metrics: frame throughput, accuracy and energy, we are better able too determine trade-offs between algorithms.

With the increasing number of processor cores in today's processors, we also investigate how parallel techniques can be utilised.

We use the newly released framework, SLAMBench, which provides us a platform for the evaluation and characterisation under these three metrics in a controlled environment.

1.2 Contributions

Our primary contribution is the comparison of dense and semi-dense, sequential and parallel techniques in two leading SLAM algorithms: KinectFusion (dense and sequential) and LSD-SLAM (Semi-dense and parallel). In particular we show:

1. The algorithms' dependency on the dataset to be able to perform adequately.
2. We show that KFusion (an implementation of KinectFusion) is more robust than LSD-SLAM.
3. LSD-SLAM can perform better under the three metrics, but is susceptible to poor lighting or minimal textures in the scene.
4. The utility of parallel, asynchronous pipelines to be able to 'easily' obtain real-time performance.

As part of fulfilling our primary contributions, we further contribute:

1. An integration of LSD-SLAM into SLAMBench, by providing a deterministic, process-every-frame mode.
2. SLAMBench extended to support the TUM RGB-D dataset.

1.3 Structure

We begin by covering the required background material (Chapter 2), required to understand the two algorithms, KinectFusion and LSD-SLAM (Chapters 6, 7 respectively). We do this without reference to implementation details. We then proceed by describing the steps taken, and challenges solved, to integrate LSD-SLAM into SLAMBench (Chapters 9, 10 and 11).

In the second part, we perform a characterisation of the algorithms by using an implementation of them. We begin with KFusion (Chapter 14), and LSD-SLAM (Chapter 15), within the SLAMBench framework (Chapter 13). We delve further into LSD-SLAM with a design-space exploration (Chapter 17), including investigating performance on different hardware platforms. We close the second part, with an evaluation of how it performs outside the confines of SLAMBench (Chapter 18).

We finally summarise the highlights of investigation (Chapter 19) and conclude our investigation, including outlining future directions (Chapter 20).

Chapter 2

Background

We begin by outlining the SLAM problem. We focus on one class of methods of solving this problem, namely monocular vision, which will limit our focus of the remainder of the report. Following on, we define a set of features, which should be present in SLAM algorithms, as a the basis for a method of benchmarking a SLAM algorithm. To facilitate benchmarking, we outline, the existing 'SLAMBench' framework.

2.1 What is SLAM?

2.1.1 Conceptually

Consider being dropped into an uncharted city, and as you walk around, navigating without getting lost. Your primary problem is (not) knowing your location. One method to solve this is to create a map. Therefore, whilst moving around, recording a particular set of features of the surroundings, updating the map using your current location estimate, and simultaneously using the map to provide an updated location estimate - a cyclic dependency. By moving farther away from your initial position, the map quality will most likely degrade and therefore the affect the accuracy of the location estimate which in-turn will degrade the map further, by updating the map in a slightly wrong location. This is clearly a 'chicken and egg problem' [6].

2.1.2 Concretely

This problem of simultaneously locating and mapping the scene is known as '**S**imultaneous **L**ocation **A**nd **M**apping', *SLAM*. The idea of SLAM was born in 1986 at the IEEE Robotics and Automation Conference, where conversations were starting to take place about the problem in the context of robotics [7].

Shortly after the inception of the SLAM problem, it was determined that it can be solved iteratively. Smith *et al*, showed that when a observation is made the map or model must be updated, along with the pose (location) of the vehicle. It was first thought that this problem was divergent (i.e. the generated map increasing becomes further from

the truth), and therefore no solution could be found. However, as it is now known, it is convergent and therefore with repeated observations, a consistent map can be derived [7].

An example of a robot moving through a space, can be seen in Figure 2.1, which shows a series of observations of landmarks which it uses to *locate* itself. The absolute location of the landmarks is never recorded, as all measurements are relative to the robot at the point they were taken [7].

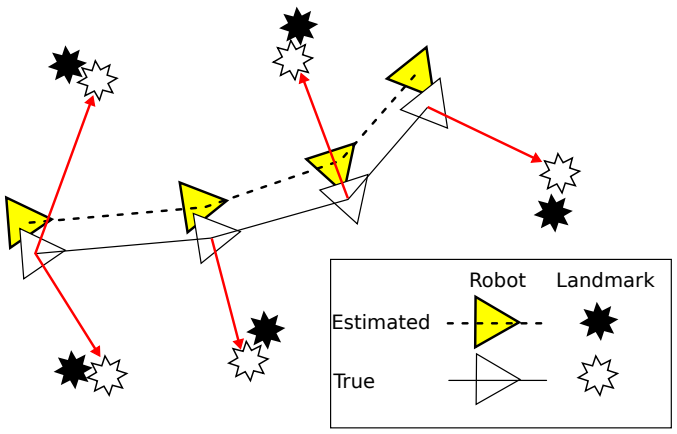


Figure 2.1: An example of taking observational measurements, specifically highlighting the differences between estimated and ground truth poses and landmark positions. Redrawn from "Simultaneous Localisation and Mapping (SLAM): Part I" [7].

Moreover, the aspects of the landmarks recorded will be very dependent on the SLAM algorithm and the available inputs. There are a variety of observational methods e.g. features, thus recording edges and/or corners of objects, or distance measurements.

2.2 Monocular SLAM Methodologies

There are many different approaches attempting to solve the SLAM problem. In this report we only consider vision based approaches, thus we can narrow our discussion. A primary concern when developing a vision system is determining what features to track between frames [6]. Through the motion of the camera, the structure of the environment (so-called Structure from Motion), can be determined and concurrently the pose can be incrementally determined and updated [8].

Vision based systems provide a rich data source to perform odometry [6]. Initially, vision based SLAM systems used stereo-pair of cameras as input, in a way similar to human vision. Stereo systems provide the ability to determine the depth of objects seen, however they require a special hardware setup and calibration. Therefore, using a single camera, 'monocular vision', can mitigate these issues (but also creates others - namely lack of scale).

Davison *et al* arguably provided the first real-time monocular vision system published in their paper "Real-time Simultaneous location and mapping with a single camera" [9].

This system extracted features from the scene, as observed by the camera, to build its sparse model. Importantly, it pioneered the ability to perform real time (30 FPS) monocular SLAM, however it was limited to smaller and simpler scenes [10].

The next big step in research was presented as "Parallel Tracking and Mapping" [11], *PTAM*. This split the processing into two distinct parts: tracking and mapping. The first step, tracking, attempts to find the pose of the camera using the generated map, which is assumed to perfect. Mapping is done in parallel, and attempts to globally optimise the map as updates from the tracking stage are inserted.

PTAM and the work by Davison, extracted features, edges or corners from the scene. A different approach, but following on from PTAM, is to use all the data available, realised as "Dense tracking and mapping" [12], *DTAM*. Tracking is performed by aligning frames to key-frames. Key-frames store depth amongst other data, within the map.

This brief history provides context but also gives us a method to classify SLAM algorithms.

2.2.1 Classification

We use two methods to classify the two algorithms used in this report, based upon this history.

Quantity of Data Used: Dense vs Semi-Dense[1]

The first categorisation, is by the amount of data the algorithm uses: dense and semi-dense. Dense methods, such as those used by DTAM, utilise all the data available in the input (e.g. all the pixels in a frame). On the other hand semi-dense methods will only utilise a subset of the data, in effect summarising the data by choosing points to follow based on some criteria. Moreover, the points tracked could be colour intensities or a feature, e.g. edge or corner.

Pipeline Structure: Sequential vs Parallel

The final category we consider is the processing pipeline structure, which could either be parallel or sequential, essentially if the algorithm follows a PTAM like approach or not.

2.3 Components of a SLAM Systems

We have seen that both PTAM and DTAM, as from their names, there is are **tracking** and **mapping** components to SLAM. At least in the approaches, and in the algorithms we will encounter.

[1] We use the term 'semi-dense' rather than non-dense or sparse, as the algorithms we will encounter are either defined as dense or semi-dense.

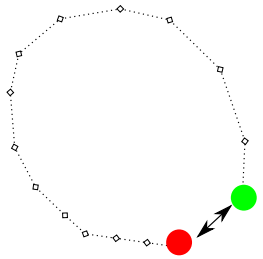


Figure 2.2: Loop-closure detection

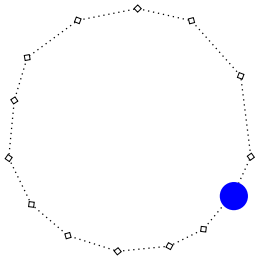


Figure 2.3: Pose update after loop-closure detection

The distinction between these two components is important, which we highlight below. Moreover, we describe another commonly found component loop closure detection which can aid and improve both tracking and mapping.

2.3.1 Tracking and Mapping

Tracking and mapping are two complementary but semi-separate components.

Firstly, tracking. Assume we have a map of the scene, as defined by the algorithm. Tracking will attempt to provide determine the current pose (globally or locally) of the sensors based on some new measurements. This though requires a suitable map, but as we have discussed previously, generally this is not known prior to the algorithm commencing and subject to sensor noise - so its not completely accurate.

Therefore, secondly, we need a mapping component. This will keep track of the observations - again determined by the algorithm - so that tracking can take place. At least in the algorithms we will encounter, to update the map, the tracking is assumed to be perfect, and the map is (incrementally) updated.

However, both tracking and mapping is subject to drift, which will result in divergent behaviour, hence the need to attempt to reduce this.

2.3.2 Loop Closures

A very important, but not necessarily required aspect of SLAM are loop closures, which can help reduce drift and improve pose estimates, by updating the map, making it more coherent [13]. This is especially important for large scenes, where the aim is to have a locally and globally consistent map.

SLAM implicitly assumes motion, which introduces a problem: drift. Returning to our conceptual example, moving away from the start location, measurements are taken of the scene and recorded. However, over time the map quality will degrade. This will be noticed when upon returning to a previously visited location, which in all probability will not show on the map, but even though physically you have returned to the same location. This is shown in Figure 2.2. By recognising (using some method) that you have returned to a location you can fix the map, Figure 2.3. In the robotics domain, this is known as loop-closure (detection) and pose optimisation respectively.

Drift

In this report, we encounter rotation, translation and scale drift. Figure 2.2, primarily shows translational drift, with rotational drift being closely linked. Scale drift is slightly different. RGB monocular vision cannot determine the precise scale of the scene, and with movement, the scale will change, therefore an algorithm not considering scale, will be scale-ambivalent. By ignoring any of these drift sources the algorithm will have it as an additional error source.

There are many ways of attacking the problem of loop-closures and thus drift, which we will highlight when we visit the algorithms. (For example in LSD-SLAM, see Section 7.3.6).

2.4 Evaluating a SLAM Algorithm

In order to evaluate a SLAM algorithm, we need to define what we consider to be a good SLAM algorithm. This can be very subjective and application specific. However, we aim to be more general, but note when particular trade-offs can be made.

2.4.1 An Ideal SLAM System

Designing a SLAM algorithm is no trivial task. A good starting point is to determine the features of an ideal SLAM system (algorithms and hardware). An ideal system will have the following properties:

- **Fast Update Frequency:** A primary assumption made by SLAM algorithms, using vision methods, is that the transformation angle and distance between the frames is small. A low update frequency means the camera can only move slowly through the scene. A high update frequency ($f \gg 30$ Hz) will mean the camera will be able to move quickly, whilst the transformation difference between the frames still be small. The system can then truly be real time[2] [14] [15].
- **Drift Free:** At any point in time the robots believed location is the correct location, and as such errors do not build up over time.
- **Accurate model of environment:** The model of the environment is accurate i.e. it is to scale and has the correct proportion [16].
- **Handle large and small environments:** The model must be able to handle any environment on any scale without prior knowledge about it.
- **Handle different lighting conditions:** Visual SLAM methods require light, clearly how the light interacts with objects and its intensity alters how the camera will view the scene.

[2] A $30Hz$ update frequency is currently considered real-time but this still limits the camera movement [14]

- **Inexpensive hardware requirements:** Many early SLAM systems required specialised and therefore expensive hardware setups. In order for robotics to become common place, commodity hardware must be used.
- **Low power requirements:** Some of the SLAM algorithms over the last few years have required powerful GPGPU's and CPU's thus having a large energy budget, hence they are not suitable for the vast number of low power applications in robotics.

All of the state-of-the-art algorithms make trade-offs of one or more of the properties outlined above. To quantitatively evaluate SLAM algorithms with respect to (some) of the above properties we require tools to aid this.

2.5 SLAMBench Framework

We now consider, the SLAMBench framework, as state-of-the-art tool for performing quantitative analysis of SLAM algorithms. To understand 'SLAMBench', we first need to understand its context, within the larger 'PAMELA' project.

2.5.1 The PAMELA Project

A **PA**noramic **V**iew of the **M**any-core **LA**ndscape - *PAMELA* - is a joint research group between Imperial College, The University of Manchester, and The University of Edinburgh, funded by the Engineering and Physical Sciences Research Council. Its aim, as outlined in its funding proposal, is to "optimise the hardware and software configurations together to address the important application domain of 3D scene understanding" [17]. This project appears to be the only research group taking "a holistic" approach to the 3D scene understanding, and attempting to discover - should one exist - the ideal SLAM system including algorithm(s). Figure 2.4 shows the various components in the PAMELA project.

2.5.2 Overview

In an attempt to understand SLAM algorithms, including how they interact with hardware, a tool, SLAMBench [16], was created. Like the PAMELA project, SLAMBench takes a holistic approach in comparing and investigating SLAM algorithms. It enables the simultaneous quantitative comparison of FPS (frames per second), accuracy and energy consumption of a particular collection of algorithms. These are the 'three SLAMBench metrics'.

1. **FPS**, how many frames can be processed per second.
2. **Accuracy** is measured as a distance, by comparing the calculated trajectory against a known trajectory.
3. **Energy consumption** is measured directly from the hardware sensors.

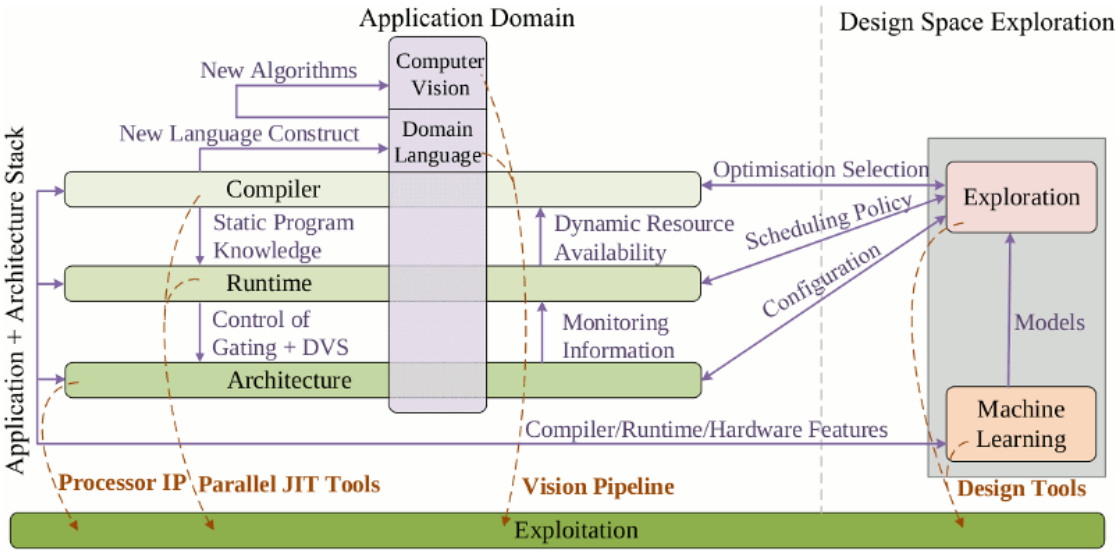


Figure 2.4: The PAMELA Project Overview [18]

Since SLAM algorithms interact with the physical world, it makes it difficult to test and compare implementations as there are so many additional variables to consider, hence SLAMBench operates in a closed, deterministic world. Figure 2.5, shows the layout of SLAMBench.

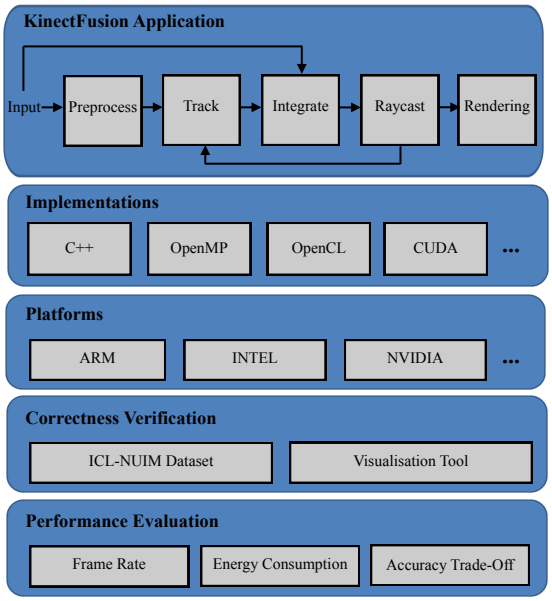


Figure 2.5: SLAMBench Framework (inc. KFusion) [16]

SLAMBench currently only works with monocular RGB-D vision SLAM algorithms, therefore making it suitable for this report. Although this may seem a limitation, there are many algorithms is this sub-domain of SLAM, with new algorithms being published frequently.

SLAMBench is not confined to a particular SLAM implementation or hardware platform. However, SLAMBench does provide a SLAM implementation based on an open-source implementation of KinectFusion (for more details on KinectFusion see Section 6), called "KFusion". Furthermore, the SLAMBench authors have provided a port of the original "KFusion", primarily in CUDA, to versions exclusively in C++, OpenMP and OpenCL. This enables a mix and match approach to investigating alternative implementations and/or algorithms and platforms.

Altogether, SLAMBench allows a scientific approach to comparing SLAM implementations or sub-part thereof [16].

2.5.3 Methodology

SLAMBench enables evaluation of SLAM algorithms under the following assumptions:

1. All frames are processed
2. Deterministic
3. Easily determine the three metrics:
   (a) Frame through-put
   (b) ATE (using the MAE)
   (c) Energy usage

The process-every-frame mode enforces that frames are not missed, otherwise the comparison will not be fair since in some algorithm architectures frames can be dropped. These combined features enable repeatable experiments, providing the ability to establish cause and effect, and rigorous algorithmic comparison.

2.5.4 Limitations

SLAMBench is currently limited in several aspects, which are outlined below.

1. Dataset Choice
SLAMBench is only as good as the dataset. Currently, the only dataset used is the ICL-NIUM Living Room trajectory 2 dataset. Moreover, any datasets used within SLAM-Bench need to have an accurate ground truth, so that an accurate drift-error can be calculated. We will be revisiting the dataset selection, frequently, within this report.

2. Does not Evaluate the Mapping Accuracy
Currently on the trajectory error is calculated. Mapping plays a crucial role in all SLAM algorithms and is therefore an important aspect to measure.

3. Power Monitoring
SLAMBench relies on the hardware platform to provide power readings, therefore they cannot be taken where there is no support. The original SLAMBench paper only performed power analysis on the ODROID XU3 [16].

We address the first and third issues within this report. The second is left for future work.

2.6 Summary

We have briefly outlined the SLAM problem and features found in an ideal SLAM algorithm. This led to the introduction of SLAMBench, which facilitates SLAM algorithm comparison, and provides the framework used in our SLAM algorithm comparison.

Chapter 3

Algorithm Analysis

We have just discussed at a high-level the features and the framework, 'SLAMBench', which enable us to evaluate a SLAM algorithm.

We will now precisely define how we will characterise the SLAM algorithms which we will encounter. We begin with the traditional asymptotic analysis, and move onto the parallel behaviour. We conclude with precise definitions of how we evaluate a SLAM algorithm using domain specific tools.

3.1 Asymptotic Behaviour

The traditional method of describing algorithmic performance is to describe its asymptotic behaviour with respect to the input size. There are three primary notations used when characterising algorithms in this way:

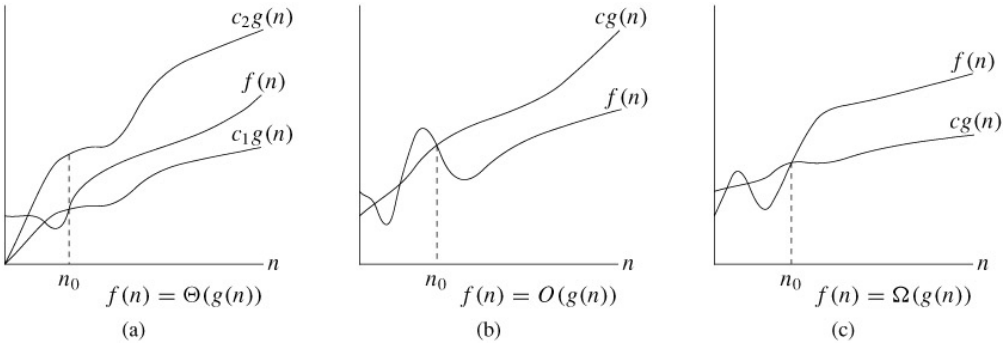


Figure 3.1: Three commonly used asymptotic analysis tools [19]. ($c, c_1, c_2 \in \mathbb{R}$ are constants)

The first, Big Theta[1], $\Theta(g(n))$, describes how a function $f(n)$ is "sandwiched" [19] between $c_1g(n)$ and $c_2g(n)$ - asymptotic upper bound. A second, Big Oh, $O(g(n)$, states $f(n)$ is bounded above by $cg(n)$ - asymptotic upper bound. Finally, $\Omega(g(n))$, states $f(n)$ is bounded below by $cg(n)$ - asymptotic lower bound.

[1] n is the input size

An asymptotic characterisation of an algorithm is independent of the implementation and is frequently used in the comparison of the run time and memory usage. The calculation of the behaviour is usual carried out by inspection and analysis.

3.2 Parallel Behaviour

The asymptotic behaviour of an algorithm is important, especially when comparing two algorithms. However, a major focus in this report is to study the parallel (and sequential behaviour) of SLAM algorithms, which the asymptotic analysis does not easily express.

We will encounter a variety of parallel algorithms, from large building blocks down to small functions ('kernels'). Therefore, as a first step, we outline parallelism, how we can analyse the algorithms we encounter. We finish with seeing how it can be realised in hardware, focusing on those features we will encounter.

3.2.1 Types of Parallelism

Parallelism can be broadly defined by two categories [20]:

- **Data Level Parallelism:** The data can be operated on in parallel.
- **Task Level Parallelism:** The tasks can be run largely independently, therefore executed simultaneously.

3.2.2 Kernels

SLAM algorithms are usually built from a large bodies of code, containing a variety of algorithms within. We therefore divide the code into smaller self-contained bodies of code, *kernels*. The purpose of extracting kernels is two fold:

1. **Simplification:** It is easier to reason about.
2. **Comparison:** By extracting kernels different implementations can be swapped and / or analysed as a single module.

3.2.3 Parallel Patterns

Using the extracted kernels, we are able to reason about the algorithm contained within. This is especially important when considering how they utilise parallelism.

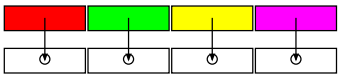
In software development, patterns are a core concept enabling ideas and methods to be reused. We determine the pattern used through inspection. This requires understanding how a kernel interacts with its data - both input and output - as well as the processing required.

In this paper we also follow the definition of parallel patterns as presented by McCool [21]. They can be broken down into two categories[2]:

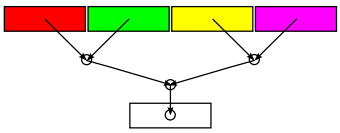
[2]This follows the same definitions used in the SLAMBench paper. Moreover, the diagrams are based upon those in the SLAMBench paper [16]

Processing

Map - Applies the same function to all elements in a collection, modifying or generating a new collection.

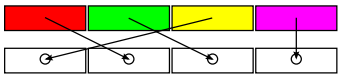


Reduce - Applies a function to all elements pairwise, reducing them to a single element.

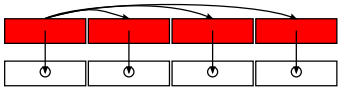


Data Management

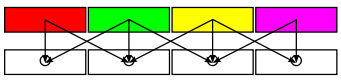
Gather - A function applied to all indices except memory access is random (implying indexable data).



Search - Retrieves data from a collection, through filtering in parallel.



Stencil - The operation acts on a spatial neighbourhood rather than a single element.



The benefit of defining algorithms by their parallel patterns is two fold: firstly, it is easier to describe and consider asymptotic behaviour; secondly, port to different parallel architectures .

3.2.4 Hardware-Level Parallelism

Parallelism can be realised in hardware in a variety of ways. It is important to consider this, as we use concrete runtime analysis, rather than just considering abstract parallelism.

Instruction Level Parallelism

Firstly, focusing on in at the instruction level, as categorised by Flynn [22], there are four board categories of instructions:

1. **SISD:** Single Instruction, Single Data.
2. **SIMD:** Single Instruction, Multiple Data.
3. **MISD:** Multiple Instruction, Single Data.
4. **MIMD:** Multiple Instructions, Multiple Data.

We primarily encounter SISD and SIMD. SIMD are realised in the architectures we use - which we define later, in Section 13.2.2 - as *Streaming SIMD Extensions* for the x86 architecture and NEON on some ARM architectures.

Multiple Processing Cores

At a higher level, still in hardware, parallelism can be achieved by having multiple processing cores, irrespective of type of instruction level parallelism. The good method to achieve multi-processor utilisation, as we see later on, is through splitting the work into multiple threads, which can be acted on in parallel.

General Purpose Graphics Processing Units

In an entirely different context, *General Purpose Graphics Processing Units*, GPGPU's exploit data-level parallelism by applying a single instruction stream to a large collection of data items, in parallel. (We do not investigate this platform, but we mention it in relation to KinectFusion.)

3.3 Characterising a SLAM Algorithm

Simply comparing the parallel behaviour does not enable us to analyse the analyse the domain specific characteristics, especially the three SLAMBench metrics. Below we define the methods allowing us to perform this quantitative analysis.

3.3.1 Accuracy

Accuracy is the foremost domain specific characterisation. There are two forms of accuracy to measure, *location* and *mapping* (clearly from the name SLAM!).

Location: A comparison checking where the algorithm believes it thought it was located, compared to where the it really was (in monocular vision this is usually the camera). It can be evaluated in two ways: *relative pose error* (RPE) and *absolute trajectory error* (ATE). RPE evaluates how expected trajectory correlates to the calculated trajectory by the algorithm, over a period of time. ATE in the other hand, compares the two trajectories at discrete time points, usually once per frame. The error is the absolute difference between the paired poses [23]. Furthermore, it has been argued that the ATE encodes the RPE, so we do not actually need to check both [23]. We use ATE in this report.

Mapping: This comparison checks how accurate the generated map is, comparing as well with a ground truth model. One can argue that if the location tracking is good, then the map must be reasonably good as well, therefore lessening the need to perform a map comparison, as getting an accurate map, for comparison, of the scene is non-trivial.

Calculating the ATE

Primarily there are two possible methods to calculating the ATE: *mean absolute error*, MAE, or *root mean square error*, RMSE. Given a ground truth trajectory G and a corresponding calculated trajectory, t, both of length n, the methods are defined as follows:

$$\text{ATE}_{\text{RMSE}} = \sqrt{\frac{\sum_{i=1}^{n} ||G_i - t_i||^2}{n}} \tag{3.1}$$

$$\text{ATE}_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^{n} ||G_i - t_i|| \tag{3.2}$$

Especially for larger errors the RMSE penalises the ATE more than the MAE which weights all magnitudes of error equally [24]. In this report we use the MAE, like the SLAMBench paper. However, simply using the MAE hides the spread of the data, which the RMSE encodes.

3.3.2 Energy Consumption

A more recently considered characterisation of a SLAM algorithm is its energy consumption. For vision based robotics an electrical power source is required, to power both the CPU and the camera.

Robots are frequently mobile and as such cannot be tethered to a power source, e.g. mains electrical sources. Therefore, they need an on-board power source, with the most common being batteries. Currently, they are not an ideal power source, as they have very low power capacity, and therefore need recharging frequently. Moreover, battery capacity is not increasing at a fast rate. Figure 3.2, compares battery capacity with other technologies.

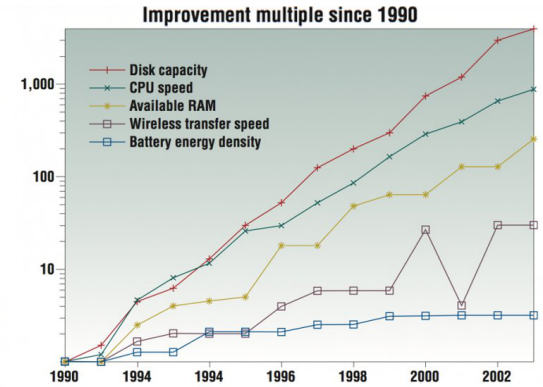


Figure 3.2: Battery capacity trend compared to other technologies [25]

Therefore, the rate of consumption of energy is critical to extending the time between recharges. Characterising energy usage depends both on the software and hardware, as

both can influence it. This is unlike the asymptotic behaviour of the algorithm, which is independent of the implementation. Moreover, peak consumption is necessary to measure as well, as a device cannot draw more power than is available.

The first step to investigating the energy usage is to determine where it is being used, the "Energy Budget". The second, and the most difficult step, is to reduce the energy usage. There are different approaches to reducing the energy usage, these include improving the algorithm, switching off unused parts of the CPU or reducing the voltage/frequency (DVFS) [26].

3.3.3 Real-Time Behaviour

An important aspect of SLAM systems is the real-time behaviour. The robot must respond to the input from its sensor(s) as a suitable rate. If its too slow, the robot has the potential to get lost.

To analyse the real-time behaviour of a system, the precise timings must be measured. Even if one algorithm is asymptotically better than another, it might still not be fast enough to meet the deadline imposed by the system.

3.4 Summary

We have seen the necessary tools and methods to characterise a SLAM algorithm, through the asymptotic behaviour, parallel patterns. We also discussed how parallel behaviour can be realised in hardware.

Finally we finished, by defining the domain specific characterisation, using the SLAMBench metrics: ATE, energy and frame rate.

Chapter 4

Preliminaries

We have seen the tools and defined how we will characterise a SLAM algorithm. However, to fully characterise a SLAM algorithm, we require a solid appreciation for its mechanics. At the core many SLAM algorithms is a mathematical foundation, which utilises a variety of models, tools and techniques. We begin by defining two tools: 'the pin hole camera model' and the 'pose representation'. Then, for the majority of the chapter we describe mathematical optimisation, particularity focusing on Least-Squares optimisation methods which we later encounter in a variety of places and forms. We conclude the chapter, by mentioning some commonly used tricks, and a few notes regarding the remainder of the report.

4.1 Camera Model

Vision techniques, in SLAM, rely on a steady stream of images, a camera provides this functionality. A camera will capture one or more properties of the point $\boldsymbol{X}$ which has been projected onto the image plane, e.g. colour (RGB) or depth. Modern, digital cameras will provide a stream of frames, at approximately 30 Hz. The resolution of the image plane, varies, but can be for example 640×480 pixels in the case of the Microsoft XBox Kinect [27].

The Pinhole Camera

Frames are a projected or 2D, snapshot of the environment as viewed by the camera. We need to be able to model this mapping to be able to 'reverse it'. The most common model is the 'Pinhole Camera Model' [28] [29]. Figure 4.1 shows a pinhole camera.

The focal length f_x, f_y (x and y axis respectively) 'scales' point from the scene to camera. Moreover, the camera centre, C does not necessarily need to reside in the centre of the image plane. The offset can then be described by (c_x, c_y), (again x and y axis respectively). This camera calibration data, or transformation is usually stored in a matrix K [28]:

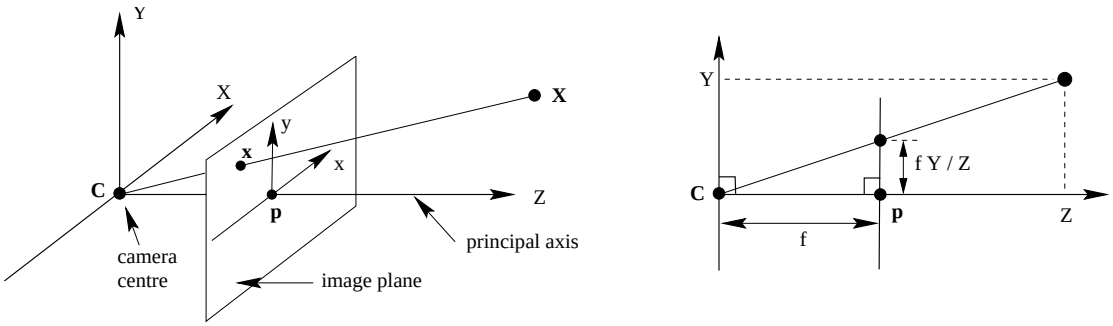


Figure 4.1: The pinhole camera geometric construction [28]

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

Assume a point $\boldsymbol{X}$ in the scene (in homogeneous coordinates),

$$\boldsymbol{X} = (X, Y, Z, 1)^T \tag{4.2}$$

then we can locate its position on the image plane,

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \mapsto \left[\begin{array}{c|c} \boldsymbol{K} & \boldsymbol{0} \end{array} \right] \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} \tag{4.3}$$

$$\mapsto \begin{pmatrix} f_x X + c_x Z \\ f_y Y + c_y Z \\ Z \end{pmatrix} \tag{4.4}$$

Importantly, the camera centre $\boldsymbol{C}$ will have a pose - translation and rotation - relative to some world coordinates.

4.2 Representing Pose

At the core of SLAM algorithms is the requirement to store and calculate pose - commonly the translation and rotation - of the robot (frequently the camera in monocular vision robotics). Firstly, if we assume a rigid body, then there are only six degrees of freedom: (**translation:** forwards/backwards, left/right, up/down; **rotation about:** x, y, z axes). More than just storing the transformations, calculations such as composition (i.e. apply one transformation after another) or inversion are required.

A common method [30] to describe the transformation T [31] is to use the Special Euclidean Group: $\mathbf{SE}(3)$ [9].

The $\mathbf{SE}(3)$ group elements can be described as follows [30] [15]:

$$T = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \mathbf{0} & 1 \end{bmatrix} \in \mathbf{SE}(3) \tag{4.5}$$

where,

$$\begin{aligned} \boldsymbol{t} &= (x, y, z)^T \quad \text{is the translation matrix} \\ R &\in \mathbf{SO}(3) \quad \text{is the 3D rotation matrix} \end{aligned}$$

4.3 Mathematical Optimisation[1]

Mathematical optimisation appears in many forms in the algorithms which we will investigate. We now provide a brief introduction to this topic.

4.3.1 The Class of Problems

Mathematical optimisation attempts to solve the problems of the form:

$$\begin{gathered} \text{minimise } f(x) \\ \text{subject to } x \in \omega \end{gathered}$$

where,

$$\omega : \text{ feasible set}$$

The feasible set is the set of valid points belonging to $\mathbb{R}^n$, for some n.

4.3.2 General Optimisation Solutions

Generally speaking, optimisation algorithms produce a sequence of points which converge to a minimum - should one exist and possibly starting at a 'good' point. The sequence of points, $x_k \in \mathbb{R}^n$ can be defined as follows:

$$x_{k+1} = x_k - \alpha_k d_k \tag{4.6}$$

where,

$$\begin{aligned} \alpha_k &: \text{ Step size} \\ d_k &: \text{ Direction of step} \end{aligned}$$

Different algorithms make different assumptions about the particular function they are minimising, therefore α_k and d_k are defined per algorithm.

[1] The following section is based on 'Introduction to Optimisation' [32], course notes for Imperial College's Department of Computing course 477 and Department of Electrical and Electronic Engineering's course 'Optimisation'.

Requirements for Convergence

An important consideration, these algorithms in general require a convex function (specifically a region of) in order to provide the global minimum, otherwise either they fail to converge or get stuck in a local minimum. In Figure 4.2, we show an example of a function with two minima and and a convex feasible region.

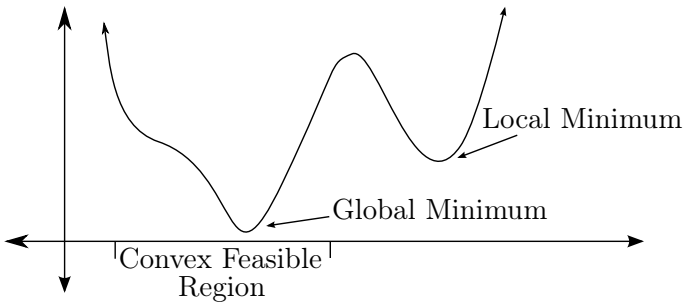


Figure 4.2: A convex graph, highlighting local and global minima.

4.3.3 Least Squares: Levenberg–Marquardt Algorithm

We will frequently encounter a specific form, known as least-squares. They are of the form:

$$\text{minimise} \sum_{i=1}^{m} r_i(\boldsymbol{x})^2 \tag{4.7}$$

where,

$\boldsymbol{x}$: A vector of parameters ($\mathbb{R}^n$)
r_i : A non linear function

Let $r_i(\boldsymbol{x}) = y_i - f(z_i, \boldsymbol{x})$, for some y, z, then one can conceptually, think of this as a problem of aligning or minimising the difference between the two 'functions': y_i and $f(z_i, \boldsymbol{x})$. r_i captures the remainder or error between the two functions. They are termed the residuals, hence the r function name.

The Levenberg–Marquardt Algorithm, is used to solve least squares problems. It is based upon the Gauss-Newton algorithm, which itself is based on Newtons algorithm. This is where we begin.

Newton's Algorithm

Newtons algorithm follows the general minimisation algorithm pattern of generating a sequence of points. To determine the direction of step, d_k, Newtons algorithm approximates the function using the Taylor's expansion, up to the second order (so making a quadratic function).

$$f(\boldsymbol{x}) \approx q(\boldsymbol{x}) = f(\boldsymbol{x}) + (\boldsymbol{x} - \boldsymbol{x}^k)^T \nabla f(\boldsymbol{x}^k) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^k)\nabla^2 f(\boldsymbol{x}^k)(\boldsymbol{x} - \boldsymbol{x}^k) \tag{4.8}$$

In optimisation, there is a first order necessary condition, *FONC* for optimality. This states, that $\nabla q(\boldsymbol{x}) = \mathbf{0}$. Hence we differentiate q:

$$\nabla q(\boldsymbol{x}) = \nabla f(\boldsymbol{x}_k) + \nabla f(\boldsymbol{x}_k)(\boldsymbol{x} - \boldsymbol{x}_k) = 0 \tag{4.9}$$

$$\boldsymbol{x}_{k+1} = \boldsymbol{x} - [\nabla^2 f(\boldsymbol{x}_k)]^{-1} \nabla f(\boldsymbol{x}_k) \tag{4.10}$$

Hence,

$$\alpha_k = 1 \tag{4.11}$$

$$d_k = [\nabla^2 f(\boldsymbol{x}_k)]^{-1} \nabla f(\boldsymbol{x}_k) \tag{4.12}$$

Gauss-Newton's Algorithm

Gauss-Newtons algorithm is a specialism of Newton's algorithm for solving non-linear least-squares problems. Defining our problem again:

$$\text{minimise} \sum_{i=1}^{n} [r_i(\boldsymbol{x})]^2 \tag{4.13}$$

where,

$\boldsymbol{x}$: A vector of parameters
n : Number of functions to minimise

Gauss-Newtons algorithm uses the same formulae for α_k and d_k, hence the first and second order derivatives of $f(\boldsymbol{x})$ need to be defined.

Define $\nabla f(\boldsymbol{x}^k)$
Let

$$\boldsymbol{r} = [r_1 \dots r_n] \tag{4.14}$$

therefore we can define the objective function (i.e. the function we wish to minimise) as:

$$f(\boldsymbol{x}) = \boldsymbol{r}^T \boldsymbol{r} \tag{4.15}$$

The gradient can therefore defined as:

$$\nabla g(\boldsymbol{x}) = [\nabla g_0(\boldsymbol{x}), \dots, \nabla g_n(\boldsymbol{x})]^T \tag{4.16}$$

where,

$$\nabla g_j(\boldsymbol{x}) = \frac{\partial f}{\partial x_j}(\boldsymbol{x}) \tag{4.17}$$

$$= 2\sum_{i=1}^{n} \boldsymbol{r}_i(\boldsymbol{x}) \frac{\partial r_i}{\partial x_j}(\boldsymbol{x}) \tag{4.18}$$

The differentials can be conveniently represented as a matrix called a Jacobian.

$$\boldsymbol{J}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial r_1(\boldsymbol{x})}{\partial x_1} & \cdots & \frac{\partial r_1(\boldsymbol{x})}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial r_n(\boldsymbol{x})}{\partial x_n} & \cdots & \frac{\partial r_n(\boldsymbol{x})}{\partial x_n} \end{bmatrix} \tag{4.19}$$

Therefore, the gradient is:

$$\nabla g(\boldsymbol{x}) = 2\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x}) \tag{4.20}$$

Define $\nabla^2 g(\boldsymbol{x})$
We follow a similar process as before:
A matrix (Hessian) of second order derivatives can be defined like the Jacobian:

$$\nabla^2 g(\boldsymbol{x}) = \boldsymbol{H}(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_1} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\boldsymbol{x})}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f(\boldsymbol{x})}{\partial x_n \partial x_n} \end{bmatrix} \tag{4.21}$$

By defining the indivual components we do not need to full calculate $\boldsymbol{H}$

$$\frac{\partial^2 f}{\partial x_k \partial x_j}(\boldsymbol{x}) = \frac{\partial}{\partial x_k}\left(\frac{\partial f}{\partial x_j}(\boldsymbol{x})\right) \tag{4.22}$$

$$= \frac{\partial}{\partial x_k}\left(\nabla g(\boldsymbol{x})_j\right) \tag{4.23}$$

$$= \frac{\partial}{\partial x_k}\left(2\sum_{i=1}^{m} \boldsymbol{r}_i(\boldsymbol{x}) \frac{\partial r_i}{\partial x_j}(\boldsymbol{x})\right) \tag{4.24}$$

using the product rule,

$$= 2\sum_{i=1}^{m}\left(\frac{\partial r_i}{\partial x_k}(\boldsymbol{x})\frac{\partial r_i}{\partial x_j}(\boldsymbol{x}) + r_i(\boldsymbol{x})\frac{\partial^2 r_i}{\partial x_k \partial x_j}(\boldsymbol{x})\right) \tag{4.25}$$

the second order derivative can be ignored above giving,

$$\nabla^2 f(\boldsymbol{x}) = 2(\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x})) \tag{4.26}$$

Combining $\nabla f(\boldsymbol{x})$ and $\nabla^2 f(\boldsymbol{x})$, we get the Gauss-Newton's algorithm step:

$$x_{k+1} = x_k - \left[2(\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x}))\right]^{-1} 2\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x}) \tag{4.27}$$

$$= x_k - \left[(\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x}))\right]^{-1} \boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x}) \tag{4.28}$$

As you can see calculating the Hessian can be avoided, it can be derived from the Jacobian, which saves having to use twice differentiable functions.

Levenberg–Marquardt's Algorithm

However by ignoring the second order derivative above, the Hessian may not be able to inverted. This will cause convergence problems, therefore Levenberg and Marquardt proposed their extension to Gauss-Newton's algorithm:

$$x_{k+1} = x_k - \left[\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x}) + \lambda diag((\boldsymbol{J}\boldsymbol{x})^T \boldsymbol{J}(\boldsymbol{x}))\right]^{-1} \boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{r}(\boldsymbol{x}) \tag{4.29}$$

The challenge in this case is to determine the value for λ. Small values of λ mean that the LM algorithm approximates Newtons', which will quickly converge if correct, but may not step in a direction of descent. Larger λ's slow the rate of convergence but should step in a descent direction. Clearly, using this method, does not guarantee convergence to a minima, however it works well in practice [33].

A Weighted Levenberg–Marquardt's Algorithm

A further, commonly used extension is incorporate weights. These weights are used to reduce the effect of outliers - this technique can only be used if there is knowledge of outliers. These could introduce some large residuals, therefore the algorithm will be optimising for these incorrect values.

The iteratively solved function is now:

$$x_{k+1} = x_k - \left[\boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{W} \boldsymbol{J}(\boldsymbol{x}) + \lambda diag((\boldsymbol{J}\boldsymbol{x})^T \boldsymbol{W} \boldsymbol{J}(\boldsymbol{x}))\right]^{-1} \boldsymbol{J}(\boldsymbol{x})^T \boldsymbol{W} \boldsymbol{r}(\boldsymbol{x}) \tag{4.30}$$

with the $\boldsymbol{W}$ defined by the implementation.

4.3.4 One Dimensional: Golden Section Method

We also encounter one dimensional problems, which have form:

$$f : \mathbb{R} \to \mathbb{R}$$

The Golden Section Method is an algorithm for solving one dimensional optimisations problems of this form, within a range $[a, b]$. (The function, f must be convex within the range.) Unlike many other optimisations techniques the function is only evaluated, the gradient or Hessian is not required. The golden section method reduces the range, $[a, b]$, by evaluating the function at specific intermediate points, so that the number of evaluations is minimised.

The details of this algorithm are not crucial to this report.

4.4 Common Tricks

There are a few interesting and frequently used tricks within SLAM, for optimisation in the mathematical and performance sense.

4.4.1 Coarse to Fine

A common trick when searching for something, is to start at a coarse level, find the (approximate) 'solution', then iterate at a finer granularity level, using the previous levels' solution as the starting point. This is useful as sometimes it can be possible to determine if the algorithm is not going to converge, therefore abandoning excess work and/or avoiding non-convergence.

This method is frequently realised when operating on an image. In a pre-processing step the image is down-sampled multiple times to form a pyramid of layers. Algorithms then start at the top layer - the coarse approximation of the real image - and proceed down the pyramid as appropriate.

4.5 Remarks

4.5.1 Reading a Kiviat Plot

We use Kiviat plots (also known as radar plots) as tool to diagrammatically show the three SLAMBench metrics: ATE, Energy, FPS. We do though however, invert the FPS, to give time per frame, as this makes it easier to compare between results. Figure 4.3 shows an example.

We can see that 'Result 3' performs the best, with 'Result 1' as the worst, with 'Result 2' performing slightly better than 'Result 1' with regards to Energy usage. As you will see if we plotted FPS this will make higher values better, where now a smaller area is better for all metrics. There is also the potential to apply weights to this, so one or more axes are more important, than the other(s).

4.5.2 Naming Conventions

In the following Sections and Chapters we use the names of parameters and variables found in the code. This is to make investigating the code, with reference to this report easier.

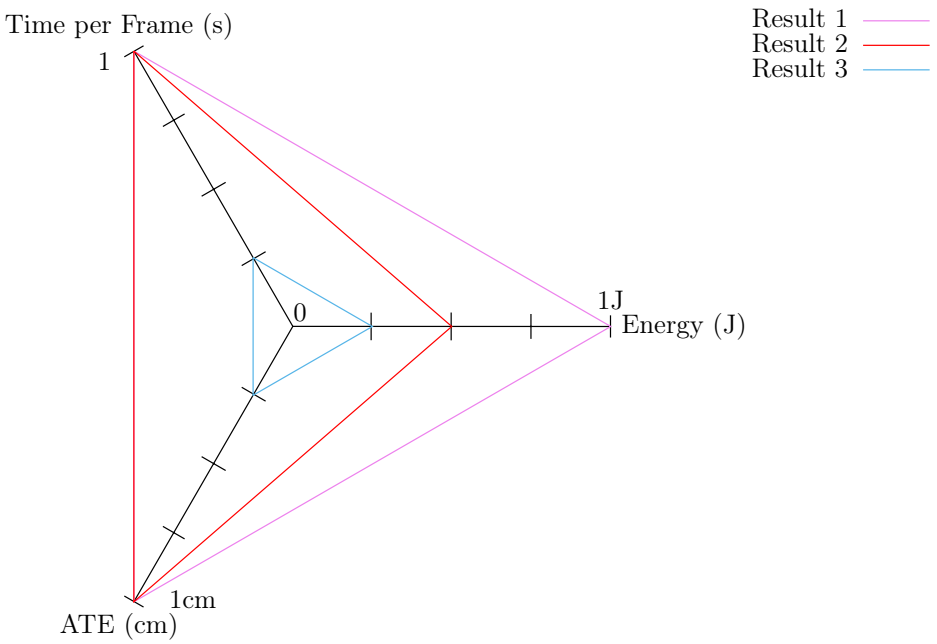


Figure 4.3: An example Kiviat Plot

Part II

Existing Algorithms

Chapter 5

Prelude

Continuing with the background to this report, we now describe the two SLAM algorithms we will characterise and compare in the remainder of the report.

We use KinectFusion, implemented as KFusion, as this already integrated into SLAMBench. We also investigate LSD-SLAM, which has recently been published.
Using our previously defined definitions, these two algorithms can be categorised as:

Algorithm	**Data Used**	**Pipeline Structure**
KinectFusion	Dense	Sequential
LSD-SLAM	Semi-Dense	Parallel

Table 5.1: The classification of the two algorithms investigated within this report.

For the remainder of this part, we first describe KinectFusion (Chapter 6) and then LSD-SLAM (Chapter 7). We do this with reference to their algorithmic structure only, with minimal reference to their implementation details.

Chapter 6

KinectFusion

Input:	1. Depth Camera
Processing:	1. Commodity CPU 2. Commodity GPU
Claims:	1. Real-time 2. Dense, accurate surface model

6.1 Overview

The Microsoft XBox Kinect Sensor, was originally designed as a new motion sensitive method for users to interact with their XBox and XBox games [34]. However, researchers realised that the Microsoft Kinect was a commodity RGB-D sensor with a suitably high frame-rate. (It provides a 640x480 RGB-D image at 30Hz [27]). They developed a method to utilise the Kinect and GPGPU (General Purpose Graphical Process Units) processors, 'graphics cards', to create a dense real-time SLAM called "KinectFusion".

Its name aptly describes its operation: it fuses the depth data from each input RGB-D frame to the scene model, (and uses the model to track the next depth frame) [9] [31].

6.2 Model Representation

Before investigating the algorithm, we need to examine its dense model of the scene - a sort of map.

In order to understand how KinectFusion represents the scene, we need to consider what it is attempting to achieve. KinectFusion aims to create a dense *surface* model of the scene, therefore the only interesting aspects of the scene are the surfaces, and that there is open space in-front of the surface. (It does not matter too much what is behind

the surface.) By using a depth camera, we are able to obtain a series (provided as a 2D 'image') of depth measurements, which, if valid, will give distances to surfaces.

6.2.1 Data Structure

The data structure used by KinectFusion to represent the scene, is to divide it up into discrete blocks, *voxels*. These are tightly packed (no overlaps, or gaps) into a rectangular prism (frequently a cube). When the prism is initialised, the dimension and resolution is defined and fixed for the duration of the algorithm. After the first frame is fused into the scene, the volume is fixed in world space, and now the voxels now represent a real physical space.

6.2.2 Truncated Signed Distance Function

To understand what each voxel stores, we return to the purpose of KinectFusion - a dense surface representation. Therefore, we are only interested in the distance to the nearest surface - *distance function*. Moreover, since the input is noisy, the measurements gathered from the depth frame will not be precise. However, with some knowledge about the error, the real distance, d will be within, μ, of the measured distance d_m, hence $d = d_m \pm \mu$. Therefore, the 'real' surface is where $d_m = d$, which gives a notion of the sign: $d + \mu$ is in front of the surface, whilst $d - \mu$ is behind - the *Signed Distance Function*. Finally, voxels only need to store a value if they are within $\pm\mu$ from the surface - *Truncated Signed Distance Function* [9]. Figure 6.1 shows an example of the TSDF.

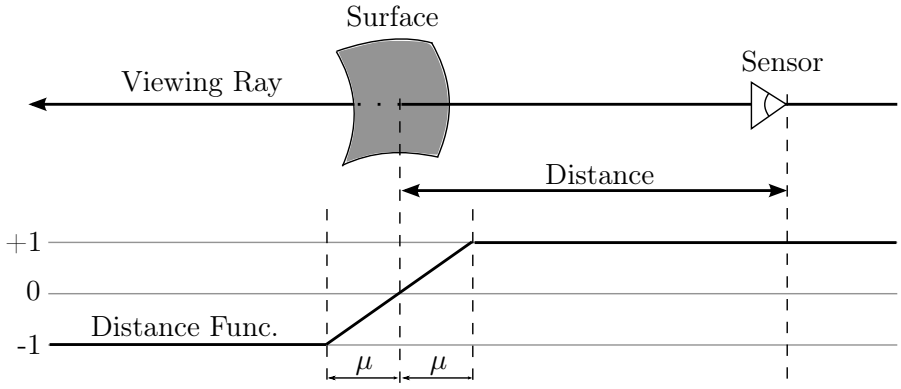


Figure 6.1: An example showing how the TSDF operates given a distance measurement between a sensor and camera. Redrawn from [35].

By repeatedly sampling the same surfaces, the noise should be able to be reduced, by averaging over many samples. To this end, the each voxel stores two values:

1) The **distance** to the nearest surface (scaled by μ) - $x \in [-1, 1]$
2) A **weight** for averaging - $w \geq 0$

An example TSDF volume can be seen in Figure 6.2.

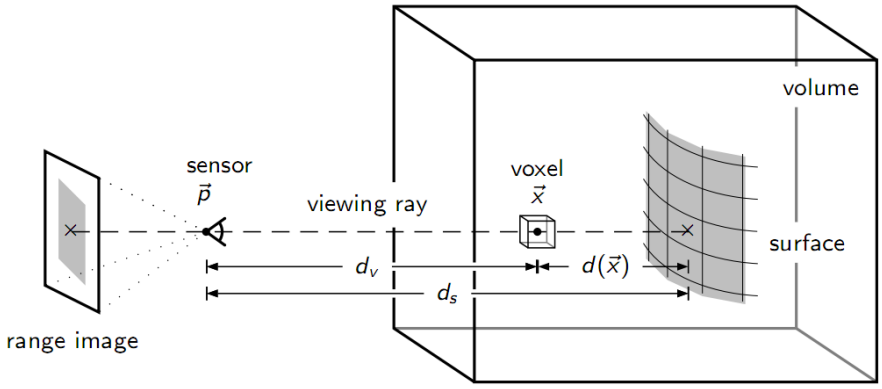


Figure 6.2: A TSDF Volume with a surface inside [35]

6.3 Algorithm

Using this insight into the model representation, we can now to proceed to understand the algorithm.

6.3.1 Overview

KinectFusion, can be decomposed into two primary components: tracking, mapping. Tracking takes a frame, and determines the pose by assuming the model (TSDF volume) is perfect. Then using the pose, updating / correcting the model, by assuming the calculated pose is perfect.

This can be further decomposed into the following high level steps, as shown diagrammatically in Figure 6.3, once per frame:

1. **Tracking**
    a) Depth Map Conversion
    b) Camera Tracking
2. **Mapping**
    c) Volumetric Integration
    d) Ray-casting

We now investigate each of these four steps in turn.

6.3.2 Depth Map Conversion

The first step is to collect and pre-process the latest RGB-D frame from the camera. As previously mentioned, only the depth data is used, so the RGB data is discarded. As with most sensors, there is noise (either (slightly) incorrect or missing data), therefore

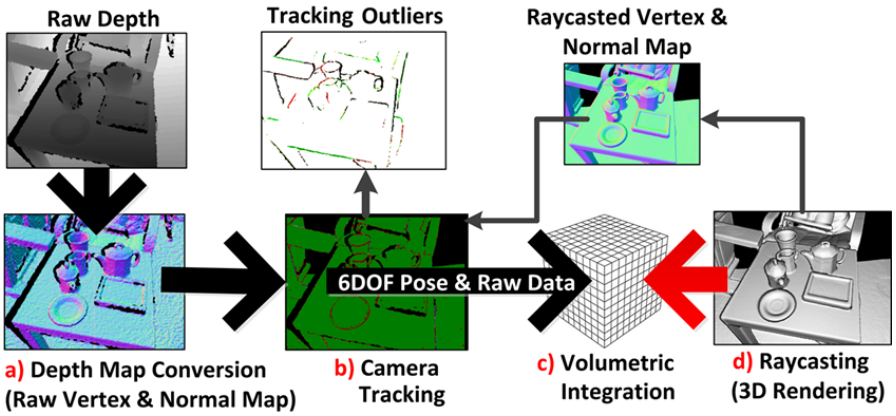


Figure 6.3: Primary steps, taken once per frame, within the KinectFusion algorithm [31]

a filter is required to remove or reduce this noise. KinectFusion uses a bilateral filter to achieve this, as it is an edge persevering filter.

The edge persevering feature is crucial as the only information stored is depth, therefore differences in depth from the camera is the only way to create a good model [36]. Figure 6.4 shows a synthesised example of bilateral filtering.

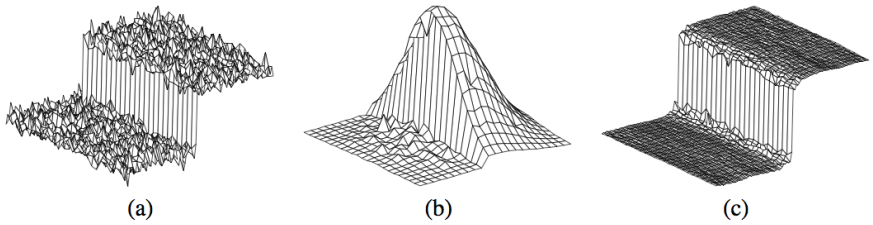


Figure 6.4: Bilateral Filter image example. The original image, with a step (a). (b) shows the combined filtering weights. (c) shows the result (n.b. step is still present) [36].

6.3.3 Camera Tracking

Camera tracking, is one of the most important steps, as it calculates the pose, of the camera frame by frame. This enables the latest depth map to be fused (merged) at the correct place in the scene model, thus improving the model and tracking accuracy. The camera tracking assumes the model, stored in the volume, is correct, and aligns the frame to this to determine the frame's pose.

To align a 2D depth map to the 3D model, the depth map is converted to a point cloud, then compared with the expected view, also a point cloud, derived from the model. KinectFusion uses an existing algorithm, called "Iterative Closest Point" *ICP*, created by Besl and McKay, and separately by Chen and Medioni [37]. ICP, utilises traditional mathematical optimisation techniques in order to align two point clouds, which provides

the relative transformation between the two point clouds, hence the frames pose can be derived.

There are many variations of the ICP algorithm [37], the variation used by KinectFusion is detailed below.

KinectFusion ICP

To generate the point cloud, the first step is to generate a down-sampled pyramid (a trick mentioned in Section 4.4). The base is the result from the bilateral filtering step. This is then down-sampled creating a new layer, progressing up the pyramid, repeating for a fixed number of times. In addition to this, each layer is converted to a point cloud, by using the inverse camera calibration matrix and the pixel's coordinates (x, y) along with the depth value.

Therefore, the top layer is a coarse version of the base layer, but still representative of it. In a simplified view, the ICP algorithm starts at the top of the pyramid using the coarser meshes to calculate a pose, which provides good approximation of the pose for the, finer layer below. This reduces the the possibility that the optimisation might get stuck in a local, incorrect, minimum. (The following explanation is inspired from [38] [39]).

A 2D example of the point comparison is shown in Figure 6.5.

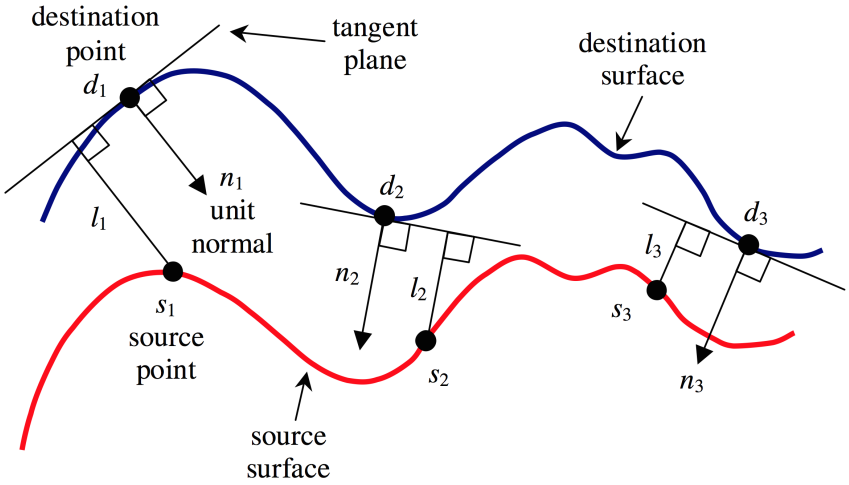


Figure 6.5: A 2D comparison of points, as it would be in the ICP algorithm [39]

We decompose the ICP algorithm into 5 steps. The first, step 0, is the initialisation performed once per input frame. Steps 1 to 4 are performed multiple times and terminate based on the criteria outlined below.

0. Initialisation

The initialisation step performs the following tasks:

1. Using camera pose from the previous time step, a projected point cloud P is created from the TSDF model.

2. The transformation matrix, $\boldsymbol{T}$ is also initialised with the camera pose from the previous time step.

1. Point Correspondence
In this first step, a map between points from $p_j \in P$ to the current down-sampled layer $l_j \in L_i$, where p_j is the closest to l_j, needs to be determined. KinectFusion uses a special method called "Projective Data Association" [9], to solve this. Projective Data Association works by synthesising the frame that the model predicts the camera will have seen at the current time point, using the previous camera pose as the pose for creating the view. This is a suitable approximation as, under an approximation, the angle between two frames is small, so the two point clouds *should* be very similar

2. Rejection
Some point correspondences maybe unsuitable (either too far away or rotated too far), as they are outliers, so a rejection criteria is used.

3. Pose Estimation by Optimisation
Pose optimisation is the core of the algorithm, as this step improves the estimate of the pose, $\boldsymbol{T}$. The aim is to minimise the distance between corresponding points by altering the transformation, $\boldsymbol{T}$. Figure 6.5 graphically shows the comparison in Equation 6.1.

$$\boldsymbol{T}_{opt} = \text{argmin} \sum_{\substack{\boldsymbol{u} \\ \boldsymbol{D}_k(\boldsymbol{u})>0}} ((\boldsymbol{T} \cdot \boldsymbol{v}(\boldsymbol{u}) - \boldsymbol{v}_k(\boldsymbol{u})) \cdot \boldsymbol{n}_k(\boldsymbol{u}))^2 \tag{6.1}$$

where,

k : Identifies a particular depth measurement
$\boldsymbol{v}(\boldsymbol{u})$: The vector at the point $\boldsymbol{u}$ in L_i
$\boldsymbol{v}_k(\boldsymbol{u})$: The vector at the point $\boldsymbol{u}$ in the model
$\boldsymbol{n}_k(\boldsymbol{u})$: The normal at the point $\boldsymbol{u}$u in the model
$\boldsymbol{D}(\boldsymbol{u})$: The depth at the point $\boldsymbol{u}$

This optimisation problem is solved *iteratively*, using Singular Value Decomposition, SVD[1] at each iteration. The number of iterations is defined by either a maximum per-level constant or if the change in angle is small enough the algorithm terminates.

Finally for this step, the transformation matrix is updated: $\boldsymbol{T} = \boldsymbol{T}_{opt}$.

4. Setup for the Next Iteration
If there is a another finer layer, it is selected and the algorithm is re-run, using the current pose as the starting point. Otherwise, the algorithm terminates.

[1]SVD is not a 'optimisation algorithm'. However, here under an assumption the transformation angle is small, this problem can be solved using SVD [39]. The details are not important for this report.

6.3.4 Volumetric Integration

After determining the pose, the depth map can be fused into the TSDF volume, as the best (but not necessarily correct) location has now been determined.

As mentioned in Section 6.2.2, each voxel in the TSDF volume stores two parameters: the *truncated signed distance value* (TSDF); and a weight. The weight is used as a running average, in order to allow the correct distance to be incrementally determined.

The integration process follows, for every voxel in the volume[2] (assume we are at time, t):

1) Select Voxel, at location $\boldsymbol{p}$:

$d_{t-1}(\boldsymbol{p})$	Distance to surface in voxel
$w_{t-1}(\boldsymbol{p})$	Weight of voxel

2) Back project onto depth image (may not be possible):

$$\boldsymbol{x} = \pi(\boldsymbol{K}\boldsymbol{T}^{-1})$$

3) Calculate the distance between voxel and the point x in the image:

$$d = ||\boldsymbol{t} - \boldsymbol{v}||$$

4) d is in voxel units, convert it to depth units:

$$\lambda = ||\boldsymbol{K}^{-1}x||$$

5) Now calculate difference between measured depth and where voxel is:

$$\delta D = \lambda^{-1}d - \text{DepthFrame}(x)$$

6) If $\delta D > -\mu$, update the voxel:

$$w_t(\boldsymbol{p}) = w_{t-1}(\boldsymbol{p}) + 1$$
$$d_t(\boldsymbol{p}) = \frac{w_{t-1}(\boldsymbol{p})(min(1, \delta D)sign(\delta D)) + w_{t-1}(\boldsymbol{p})d_{t-1}(\boldsymbol{p})}{w_{t-1}(\boldsymbol{p}) + w_t(\boldsymbol{p})}$$

6.3.5 Ray Casting

The final step, is ray-casting. This is performed for two reasons: primarily for rendering the view used for tracking, as we mentioned in the ICP algorithm. The second reason is to allow a view of the scene to be shown to the user or manipulated by other algorithms. Traditional methods of ray-casting can be used, but care needs to taken, as there are a few cases to note. The zero crossing point, from positive to negative values, is an object edge. However crossing from negative to positive or exiting the model, is caused by a lack of data and therefore is an indeterminate result [9].

[2]This is based on the KFusion code in SLAMBench and the KinectFusion publication [31]

6.3.6 Bootstrapping

The algorithm requires some bootstrapping[3]. All set-up takes place at initialisation. The TSDF's within the volume must be initialised to a value of 0 and a weight of 0, indicating there is no belief about any surfaces. The starting pose must be set, e.g. for an unknown scene it could be the centre of the TSDF volume. Finally, as mentioned the resolution and dimensions (e.g. a volume of 4mx4mx4m, with a resolution of 4mm) must be decided.

6.4 Available Implementations

The original implementation of KinectFusion has never been publicly disclosed. There is however, a feature exposed in the Microsoft Kinect Windows SDK [40], which provides reconstruction abilities, but this is not suitable for bench marking purposes.

Based on the original paper, there have been two well known, open source implementations:

- **KFusion:** A CUDA based implementation [41].
- **kinfu:** An implementation within the Point-Clouds library, also using CUDA [42].

KFusion was selected by the SLAMBench authors as the first SLAM algorithm to be integrated. We return to KFusion we we begin the comparison work, in Chapter 14.

6.5 Summary

We have seen how KinectFusion can track (using ICP) and integrate depth frames into the model, to create a dense, surface representation of the scene. This is achieved using a dense surface model of the scene, stored in a volume of voxels containing the TSDF.

[3]The process of initialising the algorithm, to make it self sustaining

Chapter 7

LSD-SLAM

Input:	1. RGB Camera
Processing:	1. Commodity CPU
Claims:	1. Realtime (30Hz) 2. All scales (indoors and outdoors)

7.1 Overview

Large-**S**cale **D**irect **SLAM**, *LSD-SLAM*, takes an entirely different approach to SLAM compared to KinectFusion. LSD-SLAM accepts RGB frames as input, and only using a subset of these (namely key-frames) to summarise the scene. Incoming frames are tracked - hence determining the camera pose - against a previously selected key-frame, by comparing intensities of a subset of the available pixels. Since it does not utilise any depth input, it is scale ambivalent, and therefore able to work in small and large environments. However, a trade-off is made in that the scale of the environment cannot be determined [30]. Dealing with the lack of a depth input features heavily throughout the algorithm.

7.2 Preliminaries

Before describing the LSD-SLAM algorithm, we first describe some of core concepts.

7.2.1 Intensity Gradient

Definition. ***Intensity:*** *The degree or amount of some quality ... brightness. [43]*

LSD-SLAM uses the notion of an intensity gradient, to differentiate between pixels for comparison purposes. The intensity gradient is the difference in brightness between a pixel and it neighbour in a particular direction (within a frame) [44]. LSD-SLAM converts the RGB frames to monochrome, therefore the intensity is simply this value.

7.2.2 Key-Frame

The only input to LSD-SLAM are frames from an RGB camera. Certain frames, *key frames*, are chosen to represent a sub-sequence of the input frames.

Not all pixels in a key-frame have a depth value, just those with a suitable intensity gradient have a calculated inverse depth. For these pixels, the (inverse) depth is modelled using a Gaussian distribution. By selectively using pixels, this means that LSD-SLAM is non-dense, which is stylised as 'semi-dense'.

In LSD-SLAM, a key-frame K_i is defined as follows:

$$K_i = (I_i,\, D_i,\, V_i) \tag{7.1}$$

where,

$$\begin{aligned} I_i &\quad \text{is the image} \\ D_i &\quad \text{is the inverse depth} \\ V_i &\quad \text{is the inverse depth variance.} \end{aligned}$$

7.2.3 LSD-SLAM Pose Representations

LSD-SLAM stores pose in the 'traditional' format using $\mathbf{SE}(3)$, but it also uses $\mathbf{Sim}(3)$. $\mathbf{Sim}(3)$, the group of 3D similarity transformations, extends $\mathbf{SE}(3)$ to handle scale, so it encodes 7 degrees of freedom. It is defined as follows:

$$T = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{t} \\ \mathbf{0} & 1 \end{bmatrix} \in \mathbf{SE}(3) \tag{7.2}$$

where,

$$\begin{aligned} \boldsymbol{t} &= (x, y, z)^T \quad \text{is the translation matrix} \\ R &\in \mathbf{SO}(3) \quad \text{is the 3D rotation matrix} \\ s &\in \mathbb{R} \quad \text{scale} \end{aligned} \tag{7.3}$$

Calculating pose through optimisation

Mathematical optimisation is extensively used within LSD-SLAM, always operating on poses either $\xi \in \mathbf{SE}(3)$, or $\xi \in \mathbf{Sim}(3)$. We mentioned in Section 4.3.2, the general optimisation update procedure is:

$$x_{k+1} = x_k - \alpha_k d_k$$

where,

$$\begin{aligned} \alpha_k &: \text{ Step size} \\ d_k &: \text{ Direction of step} \end{aligned}$$

Applying mathematical optimisation directly on $\mathbf{SE}(3)$, or $\mathbf{Sim}(3)$ does not work, as firstly it needs to transformed to a vector in, $\mathbb{R}^n$, and more importantly the incremental update, α_k will mean the resulting matrix is not always a valid transformation [45]. Therefore a new representation for these transformations is needed, "the most elegant way to represent [...] transformations in optimisations is using a Lie group/algebra" [30]. Throughout the rest of the paper, the transformation ξ, is in the appropriate Lie Group and is only converted to $\mathbf{SE}(3)$ when required. Further discussion is not required for this report.

7.3 The Algorithm

With an appreciation of these preliminaries, we are able to proceed to describing the algorithm.

7.3.1 Overview

The LSD-SLAM algorithm can be divided into two semi-distinct components:

1. Tracking and Depth Estimation (Mapping)
2. Pose Graph Optimisation

The interplay between these two sub-components, can be seen in Figure 7.1.

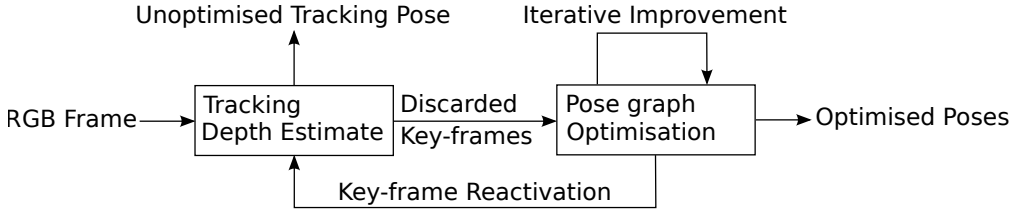


Figure 7.1: A very high-level view of LSD-SLAM decomposed into two semi-distinct components.

Tracking is performed at a local level, providing pose estimates between incoming frames and the current key-frame. Key-frames are discarded, based on a distance criteria, and are added to a 'pose graph', which, by finding loop-closures, is improved. This clearly follows a PTAM-like approach of splitting the algorithm up into distinct components, which can be executed (semi-)independently.

7.3.2 Tracking

The purpose of tracking is two fold. Firstly, to determine the pose of the latest frame, and secondly to update the current key-frame's depth-map, which is used for tracking subsequent frames.

Intuitively

Consider two frames, the key-frame and the new frame, both snapshots of the same scene from two different but very similar poses. Recall, a camera takes a 3D structure - the world - and flattens it to 2D. Therefore, with a depth map and the key-frame image we can project what the scene looks like in 3D. Therefore, using a 3D transformation we can modify these points to gain a perspective from a different view point. In our case we wish to reverse engineer the relative transformation which best describes the view we have from this new frame.

This is similar to the pose optimisation step in KinectFusion, outlined in section 6.3.3, as far as the actual view (frame) is aligned with the calculated view from the internal model.

Concretely

The latest frame is aligned to the current key-frame by minimising the photometric[1] error, $E(\xi)$. The equation describing this comparison is outlined below:

$$E(\xi) = I_{\text{key-frame}}(\boldsymbol{p}) - I_{\text{frame}}(\omega(\boldsymbol{p}, D_i(\boldsymbol{p}), \xi)) \tag{7.4}$$

where,

$$\begin{aligned} \boldsymbol{p} &: \text{The pixel position for which there is depth} \\ \omega(\boldsymbol{p}, d, \xi) &: \text{Projects } \boldsymbol{p} \text{ with depth } d \text{ onto a camera frame at } \xi \end{aligned}$$

LSD-SLAM uses the Levenberg-Marquardt algorithm to solve for the transformation ξ, outlined in Section 4.3.3. The equation which provides this, is outlined below.

$$E_p(\xi_{ji}) = \operatorname{argmin} \sum_{\boldsymbol{p} \in \Omega_{D_i}} \left\| \frac{r_p^2(\boldsymbol{p}, \xi_{ji})}{\sigma^2_{r_p(\boldsymbol{p}, \xi_{ji})}} \right\| \tag{7.5}$$

where,

$$\begin{aligned} r_p(\boldsymbol{p}, \xi_{ji}) &= I_i(\boldsymbol{p}) - I_j(\omega(\boldsymbol{p}, D_i(\boldsymbol{p}), \xi_{ji})) \\ \sigma^2_{r_p(\boldsymbol{p}, \xi_{ji})} &= 2\sigma_I^2 + \left(\frac{\partial r_p(\boldsymbol{p}, \xi_{ji})}{\partial D_i(\boldsymbol{p})} \right)^2 V_i(\boldsymbol{p}) \end{aligned}$$

[1] ...comparing the intensities of light from various sources [46].

As the depth is a an estimate, the quality of the estimate should be taken into account so that more reliable estimates contribute more the end result. The depth estimates are assumed to be Gaussian distributed [15], hence the variance term, σ^2. (We discuss this further when we describe how the depth map is generated, in Section 7.3.4.)

7.3.3 Processing Fork

The newly tracked frame, (we now have the pose, ξ) can be processed in one of two ways, depending on the 'distance' between the frame and the current key-frame.

The 'distance' is determined through two inter-frame characteristics: translation (determined from the pose) and how many points were used during tracking - in effect describing their visual similarity. A simple summed weighting of these two factors determines if a frame and key-frame are 'close' or 'far'.

For close frames - of which most are - they are used to update the depth-map of the current key-frame (see Section 7.3.4). All other frames, the 'far' ones, are promoted to key-frames (see Section 7.3.6). The distance metric between the frame and the current key-frame is defined by a weighted sum of the the translation between them and the (lack of) visual similarity between them, which is the number of depth estimates used in the tracking stage[2].

7.3.4 Depth Map Estimation Update

This stage, takes as input the 'close' frames, and is complementary to the tracking stage. Unlike the tracking stage, this stage assumes the transformation between the two frames is correct, but the depth map is not, therefore it updates the depth map. The method LSD-SLAM uses pre-dates it, but it was conceived by a similar set of authors, published in "Semi-Dense Visual Odometry for a Monocular Camera" [47].

Problem Statement

Consider a single frame, capturing a scene at a particular point in time, as shown in Figure 7.2. Clearly the distance of the point along the line is unknown, as it has been lost in the projection onto the image plane by the camera.

Assume, we have prior knowledge of the depth, modelled using a Gaussian distribution[3], this can be seen in Figure 7.3. By using a second view of the scene and epipolar geometry, we can update our knowledge, by decreasing the variance, of the depth estimates.

To understand how this can be achieved, we look at the storage structure, and the problems relation to stereo vision, before describing the solution.

[2] In the LSD-SLAM paper this is defined as a matrix, $\text{dist}(\xi_{ji}) = \xi_{ji}^T W \xi_{ji}$, but it contains only these two parameters.

[3] In general this is not required in epipolar geometry.

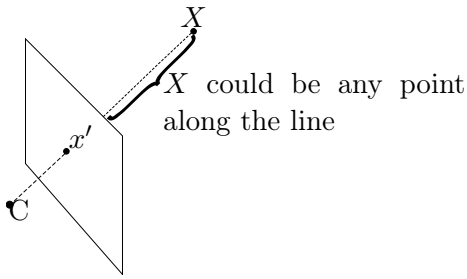


Figure 7.2: A single view of a point, X, using the pin hole camera model, with no prior information about depth.

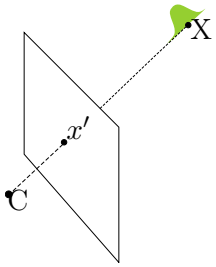


Figure 7.3: A single view of a point, X, using the pin hole camera model, with a Gaussian model representing prior belief about X's location

Storage

The depth at each pixel is stored as a Gaussian distribution, $\mathcal{N}(d, \sigma^2)$, with a mean depth, d and variance σ^2. Importantly it can be, and is frequently invalid - the reasons for which are described below. The depth at each pixel is stored as the inverse depth, for practical reasons. The depth map is stored only in the key-frames, as a 2D array.

Stereo Camera Setup

A stereo camera setup has two cameras spaced apart, sharing a common baseline, as can be seen in Figure 7.4.

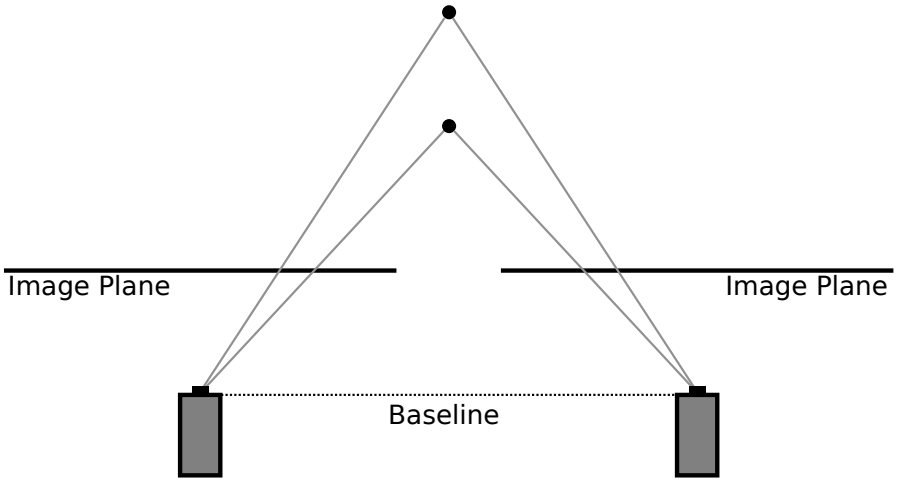


Figure 7.4: Stereo camera setup sharing a view of the scene.

However, as we have mentioned, LSD-SLAM is a monocular SLAM and therefore only requires one camera to operate. In order to update the depth map, LSD-SLAM emulates this stereo camera setup by assuming a small transformation between any two chronologically-adjacent input frames. This ensures that the two frames will take a

snapshot of a shared part of the scene, (and also that the baseline between the two cameras intersects the two frames, a necessary condition for epipolar geometry.)

Under these assumptions, a stereo camera system can be emulated, and therefore stereo vision techniques can be applied to update the depth-map [28]. Clearly, in stereo vision the relative position of the two cameras is known and fixed. However, in LSD-SLAM's case, it was calculated in the previous step, when a new frame was tracked on the current key-frame, therefore determining the transformation between them.

Updating the depth map using Epipolar Geometry

We now have necessary background to being describing the depth map update method, using epipolar geometry. We are essentially reverse engineering the projection the camera preformed, and noting how accurate we believe the depth estimates are.

For the following description, we have a key-frame x_j with many pixels having a prior depth distribution, and a frame x_{j+1}, chronologically next, tracked on x_j.

1. Baseline and Epipoles
The first step is to calculate the base-line and epipoles. The base-line is the line between the two camera centres of the frames x_j, x_{j+1}. The two epipoles are the intersection of the base line and the frame and key-frame. This can be seen in Figure 7.5. There is only one baseline between any pair of frames.

2. Pixel Selection
The next stage is to select a pixel. Select, from the key-frame, a pixel, $\boldsymbol{x}$, which is a projection of $\boldsymbol{X}$. This pixel must not have been flagged as bad[4] - in the end, all unflagged pixels will be considered. This pixel may or may not have a depth hypothesis. For the sake on demonstration, we select a pixel, with a depth hypothesis. Figure 7.6, shows such a point.

3. Search Line Construction
To update the depth estimate, we search for a visually similar pixel to $\boldsymbol{x}$ in the frame x_{j+1}. This search area is constrained, under epipolar geometry, to a particular line - the epiline - in x_{j+1}. To construct this line, we first need to construct a 'search line'. This line lies between $\boldsymbol{X}$ and the camera centre $\boldsymbol{C}$, in the key-frame x_j. This search line is constructed by inverting the camera calibration matrix $\boldsymbol{K}$, the pixel's images coordinates, and using a depth of 1. Recall the depth here is inverted hence a depth of 1 is at infinity. (Shown in Figure 7.6)

4. Pixel Search
The epiline, in the frame, x_{j+1} is parallel to this search line and starts from the epipole. We now walk along the epiline, inspecting the intensities of the pixels and their associated

[4]LSD-SLAM discards many pixels, which either have a too shallow intensity gradient or previously failed when estimating their depth.

gradients, with respect to neighbouring pixels, to find a similar pixel to $\boldsymbol{x}$, which should be a projection of the same point $\boldsymbol{X}$.

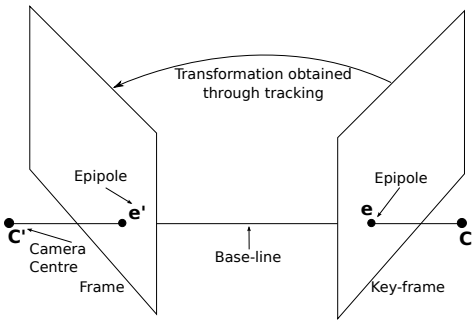


Figure 7.5: Epipolar geometry of two frames with base-line and epipoles.

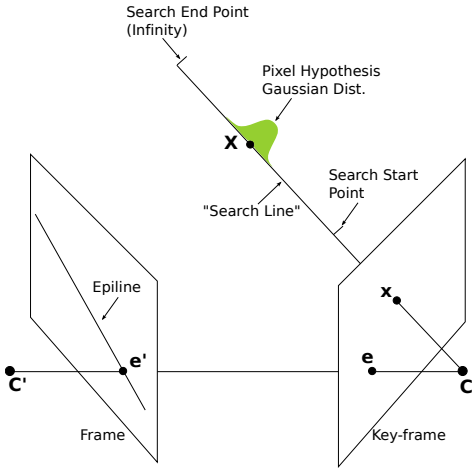


Figure 7.6: Construction of search line and epiline, for $\boldsymbol{X}$ projected into key-frame as $\boldsymbol{x}$ [28].

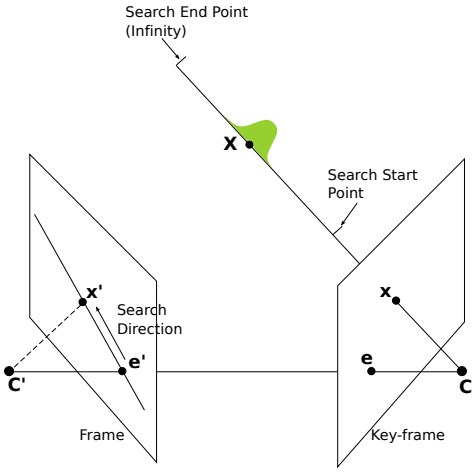


Figure 7.7: Epipolar geomtry showing epiline with an existing pixel depth hypothesis.

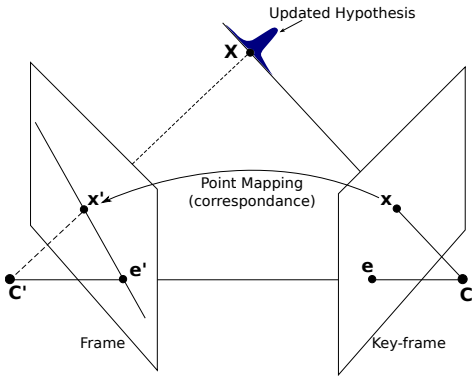


Figure 7.8: Epipolar geometry between two images, showing point $\boldsymbol{X}$ projected into both images. Point correspondence has been determined though the line search [28].

5. Mean and Variance Update

Assuming a suitable point is found, the depth estimate is updated as well as the variance. The depth is calculated from the start position hand how far down the line we have

travelled. The variance is more complicated to calculate. Engel et al. in their paper "Semi-Dense Visual Odometry for a Monocular Camera", describe how they take into account two metrics for calculating the variance. These are:

1. **Photometric:** Clearly if the intensity gradient is shallow, then an incorrect pixel maybe chosen due to noise, hence the intensity gradient is taken into consideration.
2. **Geometric:** This encodes the error in the relative transformation between the two frames.

The variance, is reduced as more measurements are taken, because the accuracy is increasing.

Once all the pixels have been considered, a sort-of 'filter' is applied to smooth the data, which is called regularisation. Its purpose is to generalise the depth-map to reduce the likelihood of over fitting [47] [48]. They also attempt to fill in the gaps (making it more dense), by using the mean of the neighbours weighted by their respective variances.

7.3.5 Frame Promotion

Recall, once a frame is tracked it can take one of two paths: update the key-frames depth map, or get itself promoted to key-frame status. This later part is what we now consider.

For frames which are too far away from the current key-frame, they are promoted to become the new key-frame. The existing key-frame's depth map is propagated to the new key-frame, by copying it over under the transformation between them.

Furthermore, for old key-frames, the depth-map is now fixed, and scaled to have a mean of one. The scale is combined with the key-frames' pose, to create a $\mathbf{Sim}(3)$ pose representation. (Recall $\mathbf{Sim}(3)$ is $\mathbf{SE}(3)$, but also including scale). These key-frames are then integrated into the pose graph.

7.3.6 Pose Graph Optimisation

Consider again, Figure 7.1, we now outline the second half of the algorithm: Pose Graph Optimisation.

The pose graph is managed by two semi-separate components: a constraint finder and the optimiser. These two components aim to improve the pose estimates of the key-frames (received from the frame promotion stage), by detecting and utilising loop-closures. They will also attempt to remove the drift (rotation, translation and scale) present in the pose estimates.

Constraints describe some connection between two frames, and are modelled using relative $\mathbf{Sim}(3)$ transformations. Frames are also described by $\mathbf{Sim}(3)$ transformations, however these transformations are absolute.

If scale were ignored, there would be two problem. Firstly, the scale would drift. Secondly, the scene could never be reconstructed since the scaling factor would not be constant at the termination of the algorithm. This would be especially true if the scene included indoor and outdoor components.

A example of LSD-SLAM's pose graph can be found in Figure 7.9.

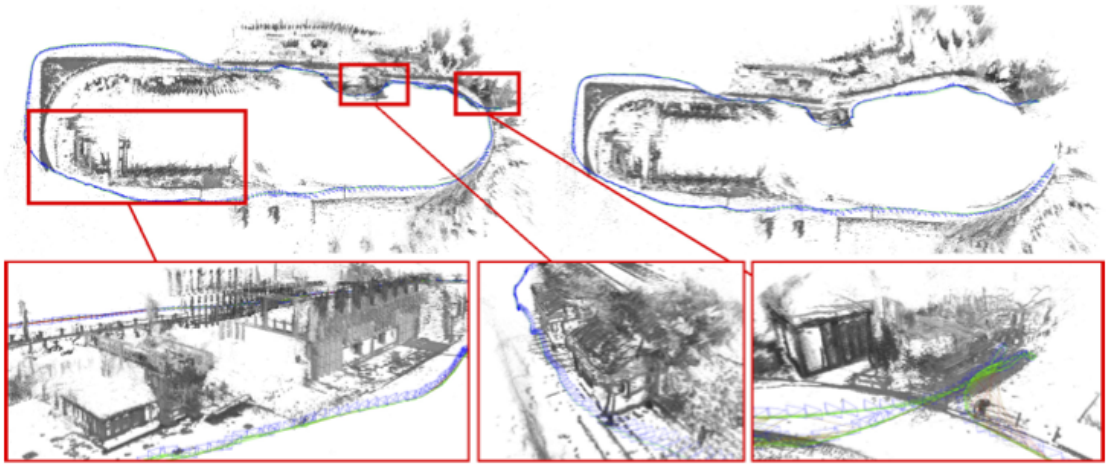


Figure 7.9: Example of a Pose Graph whilst using LSD-SLAM [15]

Overview

Assume we have collected some key-frames, and have detected some constraints between the key-frames, the pose optimisation works as follows.

The optimiser will attempt to find the best explanation of **Sim**(3) transformations which describes the current set of constraints and key-frames. LSD-SLAM, again, uses 'least squares' methods to solve this problem. To convert this pose optimisation problem to least-squares, an error metric has to be defined. Essentially it is the same as the **SE**(3) error metric defined in 7.3.2, but with a notion of scale included.

Intuitively it can be defined as:

$$E_p(\xi_{ji}) = \text{argmin} \sum_{\boldsymbol{p} \in \Omega_{D_i}} (\text{Photometric Residual under } \xi) + (\text{Depth Residual under } \xi) \tag{7.6}$$

The depth residual is the difference in depth estimates between the two depth-maps.

We explain the operation of these components starting with the data structure, then working through each component.

Constraint Generation

Given key-frame constraints are found though a series of steps. Firstly, a set of candidate frames are selected. In the paper they select the n nearest frames, "in a euclidian sense"[5] in the pose graph and those which are visibly similar. Using the combined set of frames, a reciprocal test is performed to see if the transformation between the two frames is similar in both directions to avoid adding incorrect constraints.

[5]This is configurable via the `--minusegrad` argument to LSD-SLAM.

The graph created by the constraints and key-frames can be improved over time. 'Optimised' key-frames can be re-used in the tracking stage[6], which may improve the tracking accuracy. But this only works effectively, when loop closures have been detected.

Map optimisation

Once constraints have been detected, they are added to the pose graph, and the optimiser is informed. Importantly, the optimiser only updates the key-frame poses and constraints, for frames which were not promoted they are never optimised, they just keep their relative pose to the key-frame.

7.3.7 Bootstrapping

Bootstrapping the LSD-SLAM algorithm, is a simple process, which the authors claim is suitable. Recall that each new frame is aligned to the current key-frame utilising the calculated depth map, which clearly for the first frame does not exist as their is no prior knowledge. The authors claim that randomising the depth map (with a large mean and variance for every pixel) at the point of algorithm initialisation, is suitable as the algorithm will converge to the correct depth map after a few key-frames propagations. Moreover, when there are suitable constraints the optimisation process will remove the randomisation as it converges.

7.4 Calculating the ATE

Calculating the absolute trajectory error in LSD-SLAM is not straight-forward, as we need to consider two factors:

1. Scale Ambivalence
2. Alignment

We explain each of these, then describe the solution.

7.4.1 Scale Ambivalence

LSD-SLAM is scale ambivalent, meaning it has no concept of the scene size. This is caused by using only one camera for input, and also having no depth input. Therefore, there are no ground-truth measurements taken, or which can be calculated.

So a scaled model and trajectory of a scene, e.g. a dolls house modelling a physical house, with the same trajectory should give the same result. Therefore, to the compare the calculated ATE produced by LSD-SLAM with the ground-truth, the scale needs to be determined. One method is to guess the scale but this is error prone and imprecise. Instead it can be determined by mathematical optimisation. Consider a perfect trajectory 'estimate', an example is shown in Figure 7.10.

[6] Controllable via the `--kfreactive` argument to LSD-SLAM.

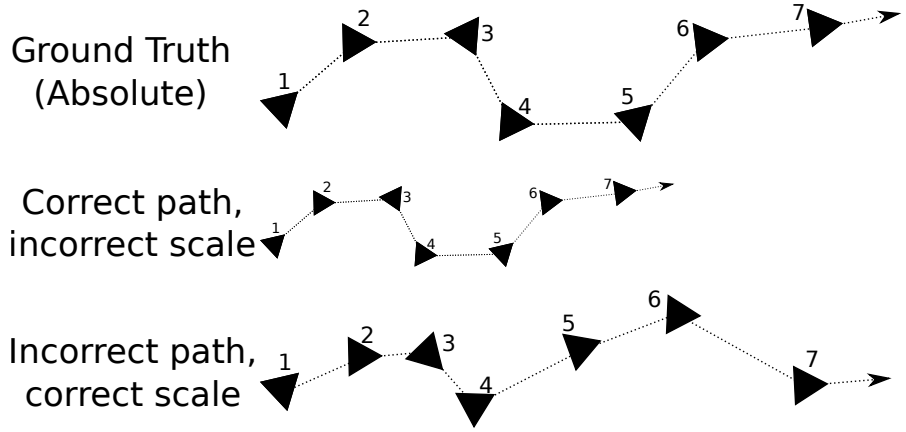


Figure 7.10: Three trajectories: Absolute ground-truth trajectory; A correct, but incorrectly scaled trajectory,; and an incorrect trajectory

The optimum ATE between the ground truth and correct but scaled trajectory, will be the one solution where the distance between corresponding points is minimised (1 to 1, 2 to 2 etc.), which there will only be one scale - a single global minimum - for which this holds. Hence this is a convex function and thus can be solved using standard mathematical optimisation techniques, such as the golden section method.

$$\text{ATE Min} = \arg\min_{scale} \text{ ATE(ground truth, results, } scale)$$

7.4.2 Alignment and the Initial Pose

Along with the scale ambivalence, LSD-SLAM does not have a fixed initial pose.

Recall, with KinectFusion, the TSDF volume is fixed in space, once the first frame has been fused. Moreover, there are no optimisation steps so the poses calculated are final.

LSD-SLAM is unlike KFsuion in that all poses are relative not absolute. This is primarily a product of the optimisation step. (You can set a starting position, but it cannot be guaranteed to be the same at the end.) Therefore, this causes an alignment problem. Clearly a perfect trajectory, ill aligned will result in a poor ATE. Therefore, any offset (rotation and translation) must be removed, before calculating the ATE. An example of an incorrect scaling and alignment (i.e. the raw result) is shown in Figure 15.9.

7.4.3 Solution

This is solved in a script provided with the TUM RGB-D collection, `evaluate_ate.py`, for calculating the ATE. Before determining the ATE, an alignment algorithm is performed, `evaluate_ate.py` uses an algorithm by Horn [49]. We have extended this script

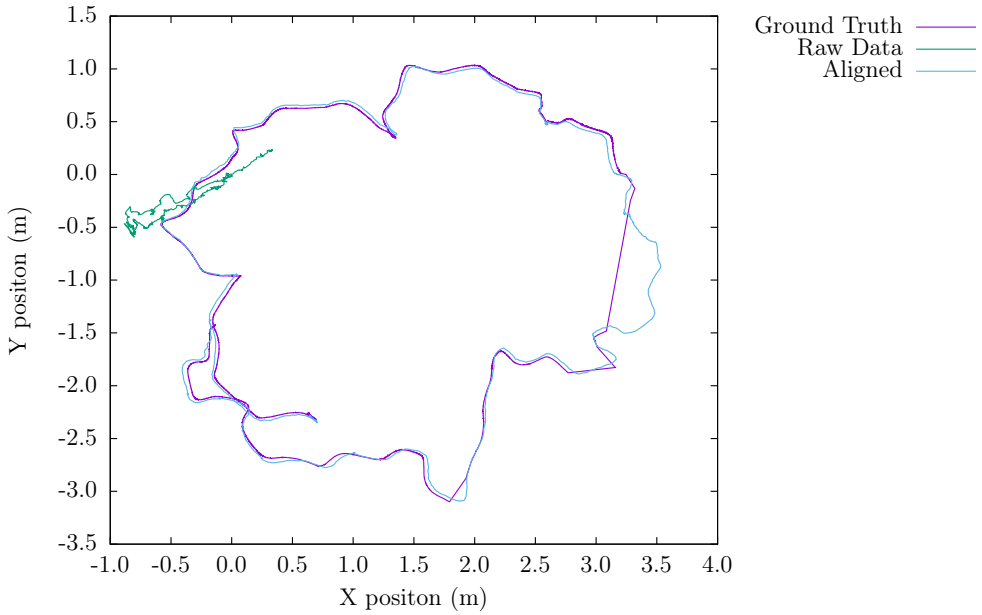


Figure 7.11: LSD-SLAM using TUM RGB-D fr2/desk to highlight need for alignment of output trajectory. Shows ground-truth trajectory, unscaled and unaligned trajectory, and scaled and aligned trajectory. (Results were collected on Seyward with default parameters).

to find the optimum scale, the Golden Section Method is briefly mentioned in Section 4.3.4.

7.5 Critical Commentary

LSD-SLAM has some interesting algorithmic requirements which we now discuss.

7.5.1 Getting Good Results

The 'ReadMe' file included with the the open-source implementation (more details on this in part: III), recommends, amongst others, the follow ideas to get best results:

- The camera on the lens should have a wide angle of view (fish eye)
- Frame rate of 30Hz
- Sufficient camera translation
- For initialisation, the camera should be moved in a circle, parallel to the scene

These are quite important to obtaining good results, and therefore could be determined as restrictions on the generality of the LSD-SLAM algorithm applications.

Take for example the initialisation step of 'moving the camera in a circle' which is presumably recommended to help the initial random depth-map to quickly converge on the correct depth by reinforcement. If the movements were more translational then the convergence rate might be slower as there will be a reduced number of reused pixels, i.e. less reinforcement of depth estimates.

7.6 Summary

We have seen how LSD-SLAM performs SLAM through the use of tracking, mapping and loop-closure detection. The scene is summarised through a series of key-frames, with tracking performed through the 3D photometric alignment between a frame and the current key-frame. The depth map of the current key-frame is updated using epipolar geometry. The key-frames are formed into pose graph so that when loop-closures are detected the key-frame poses can by improved.

We finally looked at how the ATE can be calculated by using scaling and aligning to overcome the scale ambivalence and lack of a starting pose.

Part III

Integration

Chapter 8

Prelude

In this part, the first of our contributions, we discuss the integration of LSD-SLAM into SLAMBench, which forms the basis for the algorithmic comparison in Part IV.

We begin by selecting the LSD-SLAM implementation, including considering the legal implications of such an integration (Chapter 9). We turn our attention to the selected implementation, and investigate its high-level behaviour for the purpose of satisfying SLAMBench's operational requirements (Chapter 10). This leads us to adding support for the TUM RGB-D dataset collection in SLAMBench (Chapter 11).

Chapter 9

LSD-SLAM Implementation Selection

In this Chapter we describe our selection of the LSD-SLAM implementation, taking into consideration how SLAMBench should operate and the licensing constraints.

9.1 Requirements for SLAMBench

In principle this step, simply requires importing the LSD-SLAM code, into the SLAMBench repository, however it is more nuanced than this. Returning to the purpose of SLAMBench, which is to enable comparison of SLAM algorithms and find common features and sub-algorithms for extraction, the integration must support this goal.

An ideal integration will house all integrated SLAM algorithms in one repository, including dependencies, so that they can be easily compared and kernels extracted and reused. Furthermore, common code will be extracted and there will be a consistent usage model to run an implementation for comparison purposes. The aim of this is to follow good software design patterns, where possible.

Moreover, as the SLAM algorithm implementation may be a snapshot, i.e. core development might take place in another repository by the algorithm authors, it must be simple to import changes and updates to the SLAMBench snapshot.

The following sections detail the integration of LSD-SLAM into SLAMBench, with the only existing algorithm being KFusion.

9.2 Implementation Selection

There are two primary implementations of LSD-SLAM available, a free and open source (FOSS) version, and a commercial version [50]. One could argue that there is a third, the version used which the paper is based upon - however since the code from the paper

is not available[1], thus making reproduction of the original results potentially difficult. (We address recreating the results in Section 15.1.1).

We have chosen the open source version so that the code can be integrated into SLAMBench and distributed to third parties, without commercial arrangements being made. However there is still an issue with the terms of the open source license.

9.2.1 Licensing[2]

In order for the the open source implementation of LSD-SLAM[3] to reside in the same repository as SLAMBench trade-offs need to be make. A requirement of integrating any SLAM algorithm into SLAMBench is to include all source code and dependencies, thus simplifying distribution and reducing the likelihood of 'dependency hell'. This is a challenge due to the different licenses used. SLAMBench is provided under the MIT License and LSD-SLAM is provided under the GNU General Public License version 3 (GNU GPL v3).

The primary difference between these two licences is the way derived works (i.e. distributing modifications) can be licensed.

The MIT license is a permissive license which provides "rights to use, copy, modify, merge, publish, distribute, sub-license, and/or sell copies of the Software" [51]. The only restriction is that the "permission notice shall be included in all copies or substantial portions of the Software" [51]. SLAMBench is distributed with an MIT license so that third parties can integrate their works into SLAMBench *and* distribute their modifications to whom they wish, but without having to make their modifications public. For this reasons the SLAMBench license cannot be changed.

The GNU GPL version 3 license is arguably far more restrictive. The licence is copy-left, which requires derived works to have the same license, so users downstream have the same rights and freedoms as those upstream. Since, the open source version of LSD-SLAM is being distributed with the source code available we have to do the same, if SLAMBench is distributed, including any modifications, all under the GNU GPL v3 license [52] [53].

Extracting common features, to create a shared library, using code under a GPL v3 license will cause the all the code to become GNU GPL v3 licensed - i.e. making SLAMBench GNU GPL v3 licensed. Therefore this is not an option, hence we decided not to extract any code from LSD-SLAM, and just created a basic wrapper for LSD-SLAM so that it could be used within the SLAMBench framework.

Importantly, by choosing this model, SLAMBench as a whole is now not MIT licensed, but individual components are licensed differently, this requires developers to be aware and adhere to the different licenses.

[1] I had an email discussion with and LSD-SLAM author, J. Engel, about reproducing the papers' results' he noted that "[the implementations] are very much not the same."

[2] I am not a lawyer, but this is my understanding of the issues.

[3] WE use LSD-SLAM to refer to both the algorithm and the implementation. We disambiguate where necessary.

Chapter 10

Supporting SLAMBench's Operational Requirements in LSD-SLAM

Now the level of integration has been decided, so as to comply with licensing, the integration needs to take place. SLAMBench has a variety of functional requirements, which need to be enabled in LSD-SLAM for a full and successful integration, we previously outlined these in Section 2.5.3. Moreover, there are a few extra requirements, which are not operational, but assist in making SLAMBench a 'nice' framework to work with:

1. Easy to install
2. Minimal external dependencies

In order to satisfy these requirements, we need to further consider the selected implementation. We investigate the LSD-SLAM implementation in order to satisfy the SLAMBench requirements, however we postpone a thorough investigation to Part IV.

10.1 Dependencies

The LSD-SLAM implementation utilises the Robot Operating System library, *ROS*, which "is a set of software libraries and tools that help build robot applications" [54]. Although this library is used, it is large and is primary supported on only one platform (Ubuntu [55]), therefore it cannot feature in SLAMBench. The removal was performed before this project began, by Thomas Wheelan. His version is used as the base LSD-SLAM implementation [56].

In addition to this there are a large number of dependencies. Some of these are included in the source code directories (e.g. Sophus[1]) however, others need to be built

[1] "C++ implementation of Lie Groups using Eigen" [57]

(e.g. Eigen[2], TooN[3]). Moreover, there is use of some patented algorithms, like those used by OpenFABMap, which potentially restricts the potential usage of LSD-SLAM.

This all makes for a complex distribution and setup, which goes against the easy setup feature of SLAMBench. It is not easy to overcome this as the dependencies are tightly integrated into the code. By way of getting over this, we have provided detailed instructions about how to install the dependences.

10.2 Context for Hardware Support

Investigating some of the publications before LSD-SLAM from the same Computer Vision Group at Technische Universität München, we can get a few insights into some of the assumptions of LSD-SLAM.

1. We have already mentioned the depth estimation and tracking was previously published by a similar set of authors to LSD-SLAM.

2. However, another publication, as a masters project by Kerl investigates 'Odometry from RGB-D Cameras for Autonomous Quadrocopters'. It uses whole image (dense techniques) to align two images to provide the visual odometry. The front-end of the tracking stage has a similar structure (as described in the thesis, however for example they do not go as far as using the Levenberg–Marquardt algorithm, just Gauss–Newton.) [60]

LSD-SLAM appears to combine these two techniques. Furthermore, author of LSD-SLAM, Jakob Engel, has declared two research interests, "Direct Vision SLAM" and "Camera based navigation of quadcopters" [61]. Under these assumptions there are two implications:

1. **Algorithmic:** The quadcopter must keep flying, so slow processing in other threads must be handled appropriately. This can be seen in the tracking thread.

2. **Hardware Support:** Traditionally quadcopters do not have on-board GPU's and as such LSD-SLAM does not utilise this processor type. (GPU's are commonly used in Computer Vision algorithms, like KinectFusion).

10.2.1 Optimisations

Following on from the hardware support, for the two supported processors(Intel x86 and ARM), there exists hand written optimisations primarily utilising the Single Instruction Multiple Data, *SIMD*, instructions to reduce the processing time. The two classes where optimisations have been applied is the `SE3Tracker` and `Sim3Tracker`, for **SE**(3) and **Sim**(3) tracking respectively. Table 10.1 outlines the implementations available.

[2] "Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms" [58]

[3] "TooN is a C++ numerics library" [59]

Implementation Name	Description
C++	Vanilla C++ implementation. Optimisations provided by the compiler.
C++ with SSE	The C++ with some hand optimised code blocks using the Intel SIMD instruction sets (SSE).
C++ with NEON	The C++ with some hand optimised code blocks using the ARM SIMD instruction set (NEON).

Table 10.1: LSD-SLAM implementations available in SLAMBench as supplied from the original LSD-SLAM implementation.

10.3 Architecture and Frame Progression

To support SLAMBench's operational requirements, we need to analyse the implementation. We do this by starting with the LSD-SLAM paper [15] which provides a high level diagram of the components within the LSD-SLAM algorithm. This can be seen in Figure 10.1. This provides a starting point to begin the investigation.

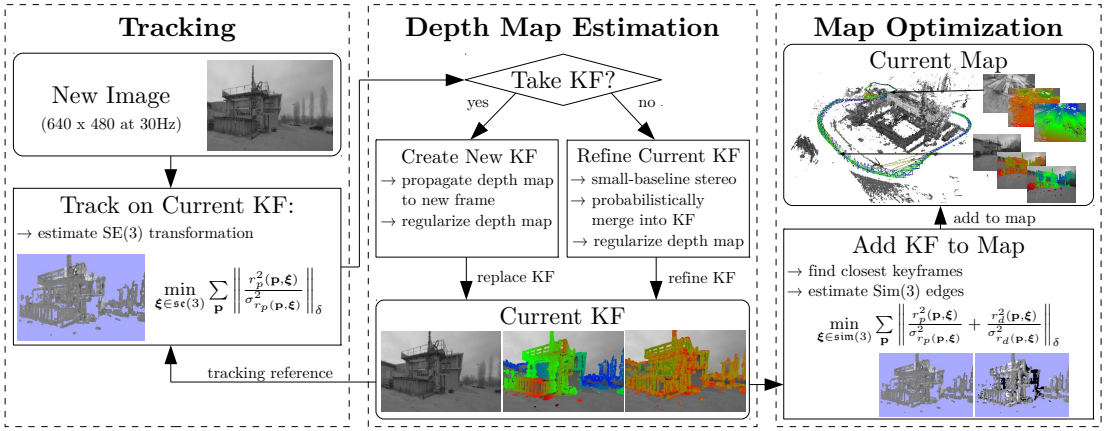


Figure 10.1: Overview of the LSD-SLAM algorithm, as shown in LSD-SLAM paper [15]

Although, the diagram in Figure 10.1 is useful, far more interesting observations can be made when investigating the code. We start by dissecting the thread behaviour (shown in Figure 10.2), which produces a similar diagram to Figure 10.1, but with inter-thread behaviour. (We will be frequently referring to this diagram and explaining its operation throughout the remainder of this report.)

However, it is worth noting now that the inter-thread communication follows an asynchronous and event driven approach. Buffers are used to hold frames ('normal' frames or key-frames) as they are passed from producer thread to consumer threads, with mutex notifications used to inform the consumer thread of buffer updates.

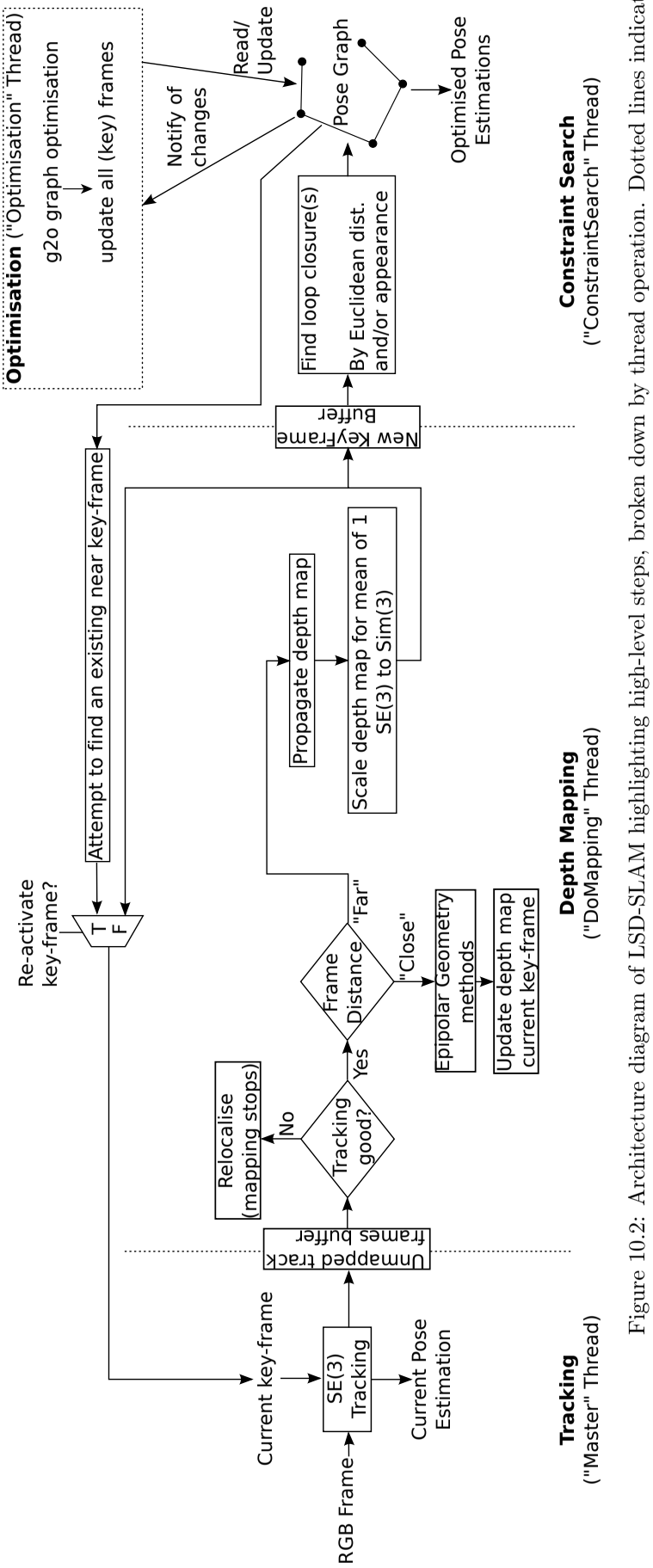


Figure 10.2: Architecture diagram of LSD-SLAM highlighting high-level steps, broken down by thread operation. Dotted lines indicate a thread's boundary of control.

Thread Block	Input	Processing	Output
Tracking	1. RGB Frame 2. Current key-frame	1. Track RGB frame on current key-frame, using **SE**(3)	1. Unoptimised frame pose 2. Tracked frame, stored in 'Unmapped tracked frames buffer'
Depth Mapping	1. Tracked Frame 2. Current key-frame	1. If frames are 'close', then update depthmap using epipolar geometry 2. If frames are 'far apart', then promote tracked frame. If reactivating attempt to reuse a key-frame	1. If 'close' • Key-frame with updated depthmap 2. If 'far' • New key-frame from tracked frame, placed in 'New key-frame buffer' • New current key-frame, either reactivated or promoted.
Constraint Search	1. New key-frame from buffer 2. Pose graph of key-frames	1. Find loop-closures by: • Distance: small translation between them • Appearance: visually similar frames (OpenFABMap)	1. Generate constraints linking key-frames, updates pose-graph
Optimisation	1. Pose-graph, of key-frames and constraints	1. Optimise - find best pose (**Sim**(3)) explanation for constraints and key-frames	1. Updated poses

Table 10.2: Explanation of processing in Architecture Diagram for LSD-SLAM in Figure 10.2

10.4 Process-Every-Frame and Deterministic Behaviour

We previously mentioned the need to enforce that all frames are processed, a so-called process-every-frame mode. In LSD-SLAM's case this is closely linked to obtaining deterministic behaviour, due to the parallel architecture of LSD-SLAM this is non trivial. We discuss the sequence of steps taken to enforce a process-every-frame mode and obtain determinism.

10.4.1 First Attempt

Initially, I believed the answer to getting a both deterministic behaviour, and a process-every-frame mode was to just consider how frames are processed between the tracking and depth mapping phase. Frames are continuously tracked under $\mathbf{SE}(3)$ and placed into a buffer ('Unmapped tracked frames' in Figure 10.2), to be consumed by the depth-mapping phase.

There is the potential for frames to be dropped from this buffer. This happens when a frame is tracked on a key-frame which is about to be placed into the pose-graph, and therefore its depth map is fixed and cannot be updated.

These dropped frames could be potentially re-processed, however for real-time applications it is best just to drop them and continue using fresh frames. Therefore, if just following this idea, to enable a process-every-frame mode is to simply wait until a frame has been processed by the depth mapping thread. This enforces a process-every-mode, however it is not deterministic, as can been seen in Figure 10.3, by the wide range of results from repeated run. Therefore, we need to investigate this further.

10.4.2 Further Investigation

Instead of just considering the first half of the processing pipeline, we now consider the behaviour of all stages. Clearly one solution is to make the pipeline sequential, i.e. choosing some sequential ordering of the parallel processing pipeline, but this will produce a new algorithm, and therefore we will not be analysing the original LSD-SLAM implementation. We therefore focus on keeping as much original behaviour as possible.

Randomisation at Initialisation

The first idea, was to remove the randomisation at the initialisation stage - recall the depth-map is initialised with random data for the first key-frame (the 'bootstrapping' phase). After some initial testing, it was clear that simply replacing calls to `rand()` by a constant does not work, therefore we have added a new parameter to the program to specify a seed for the pseudo-random number generator. *If* there are suitable loop closures, this randomisation should converge (they LSD-SLAM authors mention this in the paper), and also the poses of the first few frames will be corrected by the optimisation step. (Interestingly, the LSD-SLAM authors note that they do not fully understand why this works [15]). Fixing this randomisation problem helped, but there was still a great source of non-determinism in the implementation.

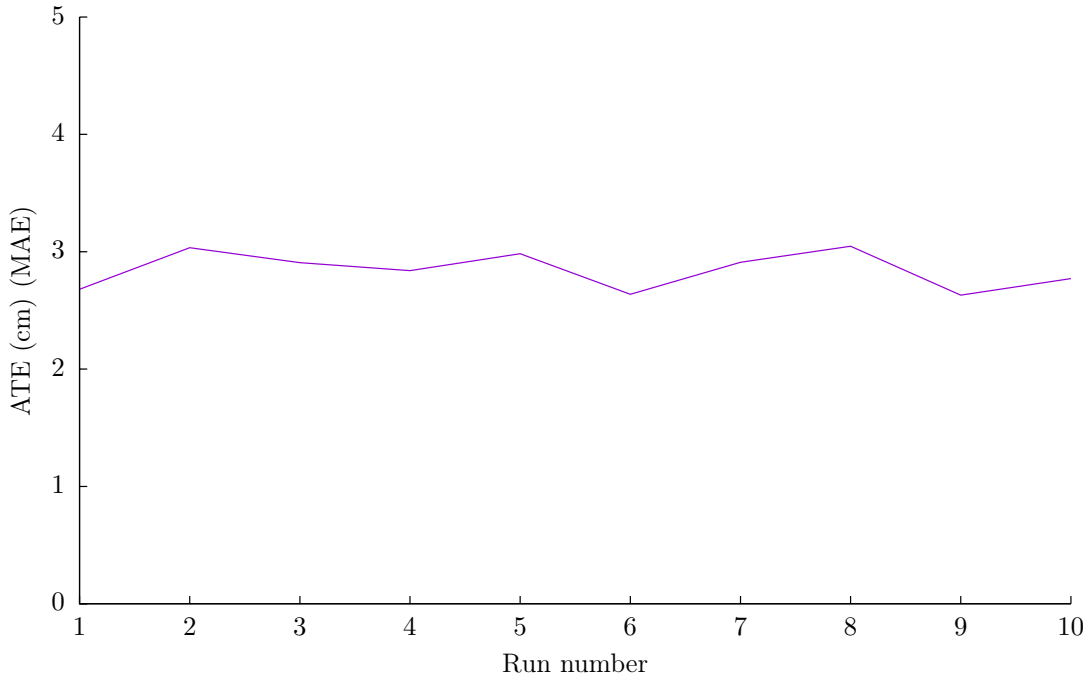


Figure 10.3: Multiple runs of LSD-SLAM using TUM RGB-D fr2/desk dataset showing varied and non-deterministic behaviour of the ATE. (Run on Seyward (x86 machine))

Non-determinism in Constraint Finding and Optimisation

After further investigation we discovered that the largest source of non-determinism is caused by the constraint generation process. The constraint generation thread finds constraints (loop closure detection) between key-frames in the pose graph, from two sources: existing key-frames which have not been visited recently, and when a new key-frame is generated. Algorithm 1 shows this behaviour.

In our experiments we have noted there there is, as expected, quite a lot of non-determinism within this thread. The primary source of non-determinism is pre-emption of the threads within LSD-SLAM. Moreover, the use of signals adds to this. To see this, in Listing 10.1, we show the actual code implementing the Line 8, in Algorithm 1.

```
boost::posix_time::milliseconds time(500);
newKeyFrameCreatedSignal.timed_wait(lock, time);
```

Listing 10.1: Sleeping in the 'Constraint Search' Thread, one of the causes of non-determinism in LSD-SLAM. This is the actual implementation of Line 8, in Algorithm 1

There are three ways this can exit: the new key-frame signal is fired due to a real new key-frame; a spurious wakeup due to implementation details of `boost::condition_variable`

Algorithm 1 Constraint Search Thread

```
1: procedure ConstraintSearchThread()
2:     while keepRunning do
3:         if new key-frame then
4:             findAndAddConstraints(key − frame)
5:         else
6:             key-frame ← random key-frame from graph
7:             findAndAddConstraints(key-frame)
8:             sleep(500ms)
9:         end if
10:    end while
11: end procedure
```

(the type of `newKeyFrameCreatedSignal`); or the timer timing out. These factors combined, mean that some key-frames can be randomly considered for constraint-search more than others.

Furthermore, the the order in which key-frames and constraints are added to the pose graph, used by g2o, effects the optimisation results, even if they are all added before the next optimisation search.

10.4.3 Solution

Before describing the chosen solution, we describe the possible solutions, to show why they are not suitable.

Considered Solutions

All of the solutions centre around the frequency and order key-frames are analysed for constraints. We considered keeping the random constraint finding of key-frames already added to the pose-graph. The possible solutions included:

1. Count the number of times a frame is considered, then making sure they are all considered the same number of times at the end.

2. Enforce an ordering on when frames are considered.

Moreover, both solutions are heavily affected by the the pre-emption of their threads. Therefore, we have not solved the problem. For example, enforcing an ordering does not mean all frames will be considered an equal number of times and enforcing an ordering will not be trivial.

Chosen Solution

The solution I chose, which is simple to implement and keeps most of the existing behaviour is as follows.

The solution is three fold. Firstly, within the constraint-search thread loop, we only consider adding constraints when a new key-frame is added. (Before LSD-SLAM reports the poses of all frames, a final constraint search and optimisation is made, for each frame, therefore we do not miss out on constraints). Secondly, we only perform an optimisation when the constraints have been added from part one. Thirdly, we sort the constraints before adding them to the g2o pose graph. Our updated version is displayed in Figure 10.4.

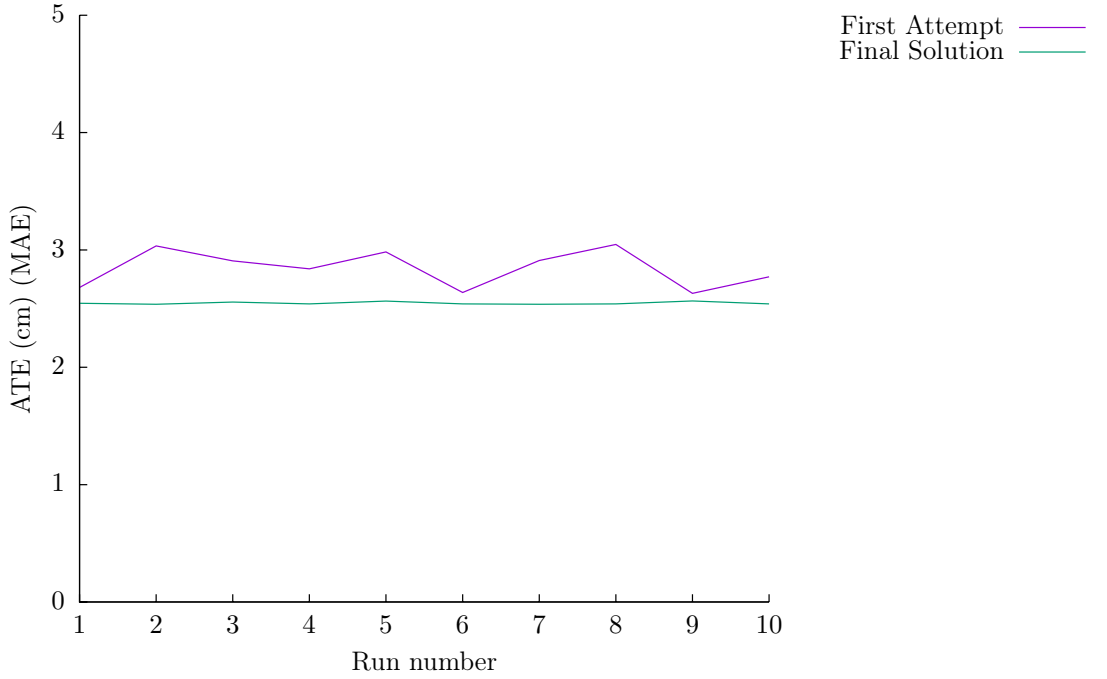


Figure 10.4: Multiple runs of LSD-SLAM using TUM RGB-D fr2/desk dataset showing the difference between first and final attempt at a deterministic process-every-frame mode. (Run on Seyward (x86 machine))

10.4.4 Critique of the Solution

We have created a deterministic setup, which is necessary for full integration into SLAMBench. However, this is limits how LSD-SLAM can behave. Firstly, this has caused LSD-SLAM to run slower, as we are enforcing an ordering in processing on the otherwise asynchronous pipeline. (But is still operates above 30 FPS, therefore it is not serious.)

More importantly, we have limited the real-life behaviour, like dropping frames between the tracking and mapping stages, if the mapping stage cannot cope with the input frequency. We address this criticism, by disabling this process-every-frame mode in Chapter 18.

10.5 Program Parameters

To conclude this chapter, we provide the parameters which can alter LSD-SLAM's behaviour.

There are a variety of parameters so that the implementation can be turned for a particular scene, shown in Table 10.4. In the subsequent chapter, we mention where these parameters are used, for additional clarity as to their meaning. Required parameters are listed in Table 10.3.

Name	**Description**	**Range**	**Default**	**Program Argument**
Camera Calibration	The file containing the camera calibration.	n/a	n/a	`--camera`
RAW file path	Path to the 'RAW' file containing the RGB-D images.	n/a	n/a	`--raw-image-file`
Log file path	Path to where the the calculated trajectory should be written	n/a	n/a	`--log`
Process Every Frame	Force each frame to be processed. Overrides FPS and blocking settings.	T / F	False	`--pef`
Frames Per Second	The number of frames added to the system per Second	≥ 0	30	`--fps`
Blocking Mode	Used in conjunction with the FPS setting so that new frames are 'released' only at the given frame rate	T / F	False	`--blocking`

Table 10.3: Required parameters for all LSD-SLAM implementations within SLAMBench.

10.6 Summary

In this chapter we have described our integration of LSD-SLAM into SLAMBench with respect to the operational requirements. We focused on how we enabled the process-every-frame mode and created deterministic behaviour. This was achieved, by waiting for the depth mapping thread to complete before tracking the next frame and enforcing an ordering and removing the random behaviour in the constraint finding and optimisation threads.

LSD-SLAM is now ready for benchmarking within SLAMBench.

Name	Description	Range	Default	Program Argument
Key Frame Distance	The euclidean distance between a frame and key-frame to determine when a new key-frame is taken. Higher value means take key-frames more frequently.	$[0, 20]$	5	`--kfdist`
Key Frame Usage	The overlap between frame and key-frame to determine when a new key-frame is taken. Higher value means take key-frames more frequently.	$[0, 20]$	5	`--kfusage`
Minimum Gradient	Minimum gradient for a pixel to even be considered for depth estimation	1 - 50	5	`--minusegrad`
Optimise Pose Graph	Find loop-closures and optimise process-every-frame	T / F	True	`--use-pose-optim`
OpenFABMap	Use OpenFABMap for appearance based constraint finding	T / F	True	`--fabmap`
Reactivate Key-frames	Reactive key-frames from pose-graph, if they are suitable.	T / F	False	`--kfreactive`
Sub-pixel stereo	Interpolate value calculating sub pixel disparity.	T / F	True	`--subpixelstereo`
Max. Loop Closure Candidates	Limit the number of candidates considered for loop closure.	$[1, 50]$	20	`--maxloopcand`
Random seed	Set the random seed for `rand()` functions	n/a	1	`--randomSeed`

Table 10.4: Behaviour altering parameters available to all LSD-SLAM implementations within SLAMBench. Extracted from those found in `https://github.com/tum-vision/lsd_slam/blob/master/lsd_slam_core/cfg/LSDParams.cfg`

Chapter 11

Supporting the ICL-NUIM and TUM Dataset Collections

Although we have integrated LSD-SLAM into SLAMBench and can potentially perform comparisons, we cannot compare directly due to the datasets supported.

The existing version of SLAMBench only supports the ICL-NUIM dataset collection. However, LSD-SLAM has only been run using datasets from the TUM RGB-D dataset collection. In order to compare the algorithms, including with the results presented in their respective papers, full support of the TUM RGB-D and ICL-NUIM dataset collections are required by KFusion and LSD-SLAM. Moreover, by providing support for the TUM RGB-D dataset collection SLAMBench, gains access to this large dataset.

11.1 Dataset differences

In order to integrate these datasets, the differences in their generation and usage need to be understood.

11.1.1 ... in generation

ICL-NUIM

The ICL-NUIM dataset was generated using a two synthetic models of a living room and office scene. These were then rendered, and noise was added to make it more realistic, which was modelled around the noise from a Microsoft Kinect Camera [62].

TUM RGB-D

The TUM RGB-D dataset collection was manufactured using a combination of a Microsoft Kinect Camera to get RGB-D frames and motion capture system to determine position. We use the TUM RGB-D 'fr2' dataset which was recoded in a warehouse [23]. There are two important issues caused by this method of dataset manufacture:

1. All the recorded data was asynchronously captured. The Microsoft Kinect Camera provides RGB and depth asynchronously, as well as the motion capture system providing the ground truth measurements. This means they are not snapshots of the scene, or location at precisely the same time. Timestamps are provided so the can be matched within some tolerance. This is unlike a synthetic dataset, like ICL-NUIM, where given a trajectory through the scene, a snapshot can be taken at a fixed point in time along the trajectory, collecting the required features, RGB frames and/or depth.

   The RGB and depth frames were captured at 30 Hz - the maximum provided by the Kinect Camera - and the trajectory at 100 Hz [23].

2. The motion capture system defines its own origin and coordinate system orientation, which is unspecified [63].

11.1.2 ... in usage

Irrespective of the frame source, frames (RGB or depth) can be inserted into the algorithm, with the poses of each being recorded for the ATE calculation.

To determine the ATE, the poses at each frame must be compared with the ground truth. Clearly, the ICL-NUIM dataset can used directly, however we encounter a problem with the TUM dataset, in that the measurements (RGB, depth, location, etc) are taken asynchronously. A script, `associate.py`, provided with the TUM dataset, provides the ability to find the closest corresponding location measurements for each frame, by comparing their respective timestamps. Moreover, due to the unknown origin, the corresponding ground truth measurements of the two trajectories (ground truth and calculated) need to be aligned. Another script is supplied, `evaluate_ate.py`, which aligns the two trajectories. (This exactly the same method which aligns the trajectories in LSD-SLAM, mentioned in Section 7.4.)

11.2 SLAMBench RAW file

The SLAMBench RAW file contains a chronological sequence of frames, as images, along with their respective sizes. This file aids the acquisition kernel, so that the system can account for the IO cost of acquiring new frames. The RAW file holds a chronological sequence of frames (RGB and depth), taken from the chosen dataset. This reduces the disk head seek time, which would not be present in a real system.

To generate this RAW file, SLAMBench provides a program to convert the ICL-NUIM datasets to RAW, namely `scene2raw`. This program chronologically inserts the RGB, depth frame pairs into the output file.

11.2.1 Producing a RAW File from TUM RGB-D

However, to include support for the TUM dataset it is not quite as trivial.

Our first attempt was to pre-associate the RGB, depth and ground truth measurements. The aim was to be able to reuse, and modify as little as possible, of the existing infrastructure, however this is not possible. This method is suitable, for the TUM RGB-D fr2/xyz dataset[1], however it is not for the TUM fr2/desk dataset. In this, with pre-association 23% of frames were being dropped, as their was not a suitable triplet of RGB, Depth a ground truth readings. This was mostly caused by a lack of ground truth location measurements suitably close - within 20 ms. (There are large gaps, on the order of seconds, in the ground truth recordings. This is odd, and does not appear to be commented on.) This produces an unsuitable RAW file as there are too many missing frames, which could seriously affect any analysis, i.e. algorithms fail to track as a result of this.

Our solution is to insert RGB frames into the RAW file in chronological order, with a suitably close depth frame. If there is not a suitably close depth frame, the RGB frame is ignored and this is reported to the user. For example, with the TUM RGB-D fr2/xyz dataset only 4 frames are removed, out of a total of 3669 frames.

This solution requires slightly different treatment compared with operating with the ICL-NUIM datasets, however this is the only reasonable solution. We have provided a set of scripts which handles and automatically processes the results (which incorporate `associate.py` and `evaluate_ate.py`).

11.3 Supporting TUM RGB-D in KFusion

A little work was required to get the TUM RGB-D dataset functional in KFusion. KFusion, along with its ATE calculation script, `checkPos.py`, assumes, that there is no rotation of the initial frame - obviously subsequent frames can rotate. This assumption is from the ICL-NUIM dataset which sets their coordinate frame 'nicely', using the right handed coordinate system, with the x, z axes parallel to the ground plane, and they have no starting rotation. However, this is not true with the TUM RGB-D datasets. We provide two ways to fix this depending on how the results are going to be processed:

1. This problem is 'automatically' solved if we align the results, for example by using the same algorithm as in LSD-SLAM (the 'Horn' method). Though this does have the potential to hide errors especially if there is a constant offset, for some reason, as this will be removed by the alignment process. (However, this is the method we use in the report. We discuss this in Section 13.3).

2. The second method, which is the one used is to supply the initial rotation of the camera of the first frame. This will remove the orientation problem. Of course there is still, possibly an offset, however this is simply removed by subtracting the location of the first frame from all ground truth measurements. (This is the original behaviour of `checkPos.py`.)

[1] We introduce the datasets properly in Section 13.2.3

Part IV

Critical Comparison

Chapter 12

Prelude

With the completed integration of LSD-SLAM into SLAMBench, along with support for the TUM RGB-D dataset, we can begin the second half of the report, the comparison. We investigate the differences of the two SLAM algorithm benchmarks: KFusion and LSD-SLAM.

We first describe the methodology (Chapter 13) which we use throughout this part. Then using this methodolgy, we characterise the algorithms, KFusion and LSD-SLAM, (Chapter 14 and Chapter 15 respectively). We then further investigate LSD-SLAM by performing some design space exploration (Chapter 17). Finally, we look at how LSD-SLAM performs outside of SLAMBench, we provides some insight into the parallel architecture benefits (Chapter 18).

Chapter 13

A Framework for Comparison

To being a comparison, we need to define our aims and methodology. Firstly, this enables us to define the terms 'better' and 'worse', but also for reproducibility reasons.

13.1 Background

In this comparison we focus on the following aspects:

1. **Characterise:** Using the methodology and ideas presented in the SLAMBench paper, we extend it to LSD-SLAM, to better understand the algorithm and parameters which affect the performance.

2. **Compare:** We compare the algorithms, chiefly with respect to the sparse and dense trade-off as well as their parallel and sequential processing pipelines. Moreover, we directly compare and contrast their data structures.

3. **Suitability for different usage cases:** Finally, we investigate their suitability primarily for mobile and desktop SLAM applications.

13.2 General Methodology

We now define the methods, tools and configuration, in order to meet the above end goals.

SLAMBench enables one to investigate the ATE, energy usage, and frame rate of SLAM algorithms. To meet this end, we take the following steps in all comparisons:

- **Pre-recorded scenes:** This enables repeatability and reduces experimental error.

- **Process Every Frame:** This together with pre-recorded scenes enables repeatable experiments and reduces non-determinism. This is especially important with LSD-SLAM with its asynchronous behaviour.

13.2.1 Simplification

In order to simplify analysis, we consider the idea of thresholds, which we define below for each metric:

FPS

Since the TUM RGB-D dataset was captured using a Microsoft Kinect Camera, which outputs data at 30 Hz, and that 30 FPS is commonly believed to be 'good' we therefore consider the handling of 30 FPS to be suitable. Moreover, handling more than 30 FPS is in some sense incorrect, if the input were a real Kinect Camera. But, investigating the peak throughput of frames gives good insight into the algorithms' performance.

ATE

There are many ways to consider what is a 'good' ATE. Firstly, just a basic minimum threshold, e.g. 2cm. This is very application dependent, so for example using the Kinect-Fusion to generate a 3D for 3D printing, one would want (sub) milli-meter accuracy, but for an autonomous car a few centi-meters are permissible. Secondly, to define it as a percentage of the scene size. Thirdly, that any value is suitable so long as tracking is maintained. We consider all these within this report.

Energy

Rather than treat energy under some threshold, we record and discuss what effect the FPS and ATE calculations have on it.

13.2.2 Hardware

In order to meet the goal 3, the "Suitability for different usage cases", we require evaluation on a higher-end desktop machine as well as an embedded platform. We use a similar set of platforms to those used in the SLAMBench paper, as that we can utilise it as a reference point. The platforms are as follows:

Machine names	**SEYWARD**	**ODROID (XU3)**
Machine type	Desktop	Embedded
CPU	i7-4770 Haswell	Exynos 5422
CPU cores	4	4 (Cortex-A15) + 4 (Cortex-A7)
CPU GHz	3.4	1.8
Language	OpenMP (& C++)	OpenMP (& C++)
Ubuntu OS (kernel)	14.04 (3.13.0)	14.04 (3.10.58)

Table 13.1: Specifications of devices used for our comparison experiments.

We choose to reuse the ODROID XU3 embedded board, as it has an interesting architecture, because it utilises the ARM big.LITTLE processor structure. A big.LITTLE processor houses two pairs of processors a lower performance set, and a higher performance set, in this case four ARM Cortex-A7 cores and four ARM Cortex-A15 cores.

13.2.3 Datasets

We use a variety of different datasets from the TUM RGB and ICL-NUIM collections. We have already highlighted their differences in Section 11, when we included support for them in the SLAMBench RAW file.

We primarily use the ICL-NUIM Living Room trajectory 2 dataset which was used in the SLAMBench paper, and two of the TUM RGB-D datasets (fr2/xyz, and fr2/desk) used the LSD-SLAM paper. We provide some details of these datasets below[1].

(For the following diagrams the X,Y axes are parallel to the floor and the Z axis is perpendicular to the floor. However, this is approximate for the TUM RGB-D datasets as we have discussed.)

ICL-NUIM

The datasets within the ICL-NUIM collection are based around two scenes: a living room and an office. The scenes are artificial, however noise was introduced based on the Microsoft Kinect Camera noise model. They each have four different trajectories within the scene [62].

[1] N.B. All the trajectories move within 3D space, we have only plotted a 2D representation above.

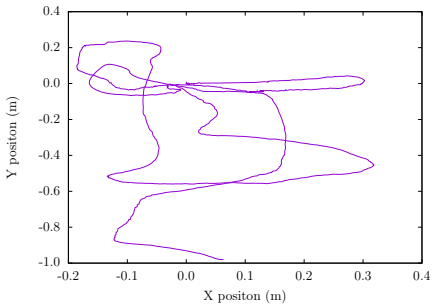
ICL-NUIM Living Room Trajectory 0

Properties:

Number of Frames: 1510

Avg. Trans. Velocity: $0.126ms^{-1}$

Trajectory



Example Frame

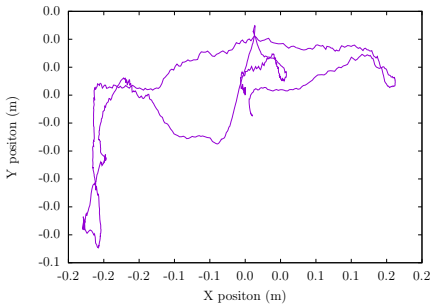
ICL-NUIM Living Room Trajectory 1

Properties:

Number of Frames: 965

Avg. Trans. Velocity: $0.063ms^{-1}$

Trajectory



Example Frame

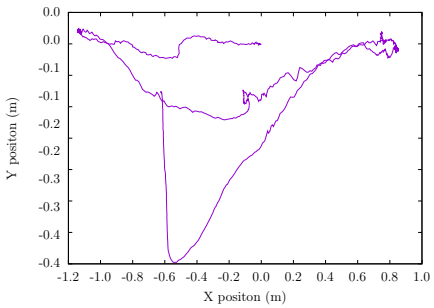
ICL-NUIM Living Room Trajectory 2

Properties:

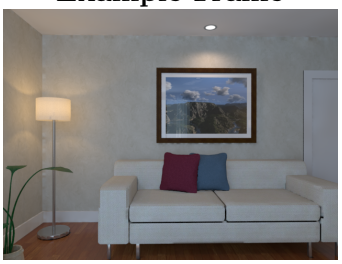
Number of Frames: 882

Avg. Trans. Velocity: $0.282ms^{-1}$

Trajectory



Example Frame



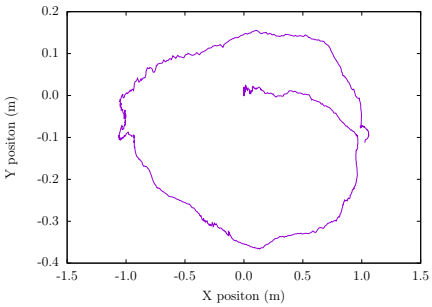
ICL-NUIM Living Room Trajectory 3

Properties:

Number of Frames: 1242

Avg. Trans. Velocity: $0.263ms^{-1}$

Trajectory



Example Frame

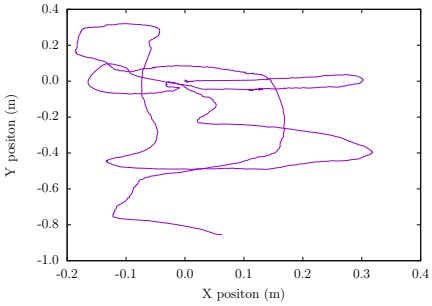
ICL-NUIM Office Trajectory 0

Properties:

Number of Frames: 1508

Avg. Trans. Velocity: $0.126ms^{-1}$

Trajectory



Example Frame

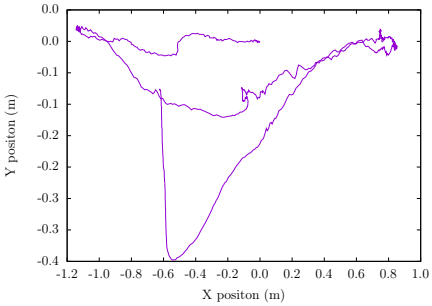
ICL-NUIM Office Trajectory 2

Properties:

Number of Frames: 882

Avg. Trans. Velocity: $0.302ms^{-1}$

Trajectory



Example Frame

TUM RGB-D

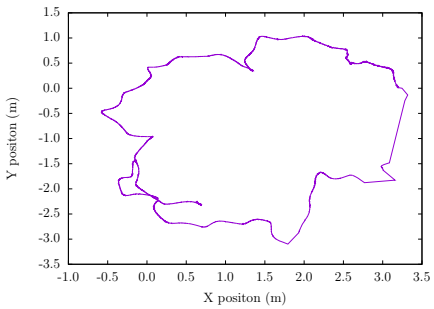
The datasets from this collection were recorded using a 'real' Microsoft Kinect Camera, in a variety of locations. We only use two from the 'fr2' dataset which were recorded in a warehouse [23].

TUM RGB-D fr2/desk

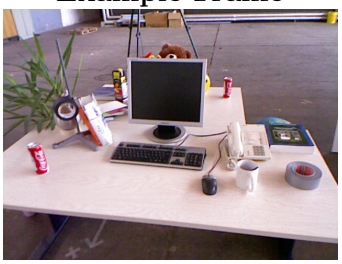
Properties:

Number of Frames:	2964
Avg. Trans. Velocity:	$0.19ms^{-1}$
Generation Method:	Microsoft Kinect Camera

Trajectory



Example Frame



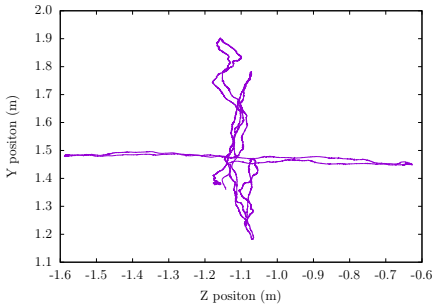
As you can see the 2D trajectory is a simple loop. However, the camera moves in the Z axis a reasonable amount, and rotates with some frames looking at the walls of the warehouse.

TUM RGB-D fr2/xyz

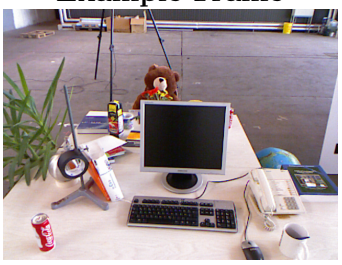
Properties:

Number of Frames:	3664
Avg. Trans. Velocity:	$0.06ms^{-1}$
Generation Method:	Microsoft Kinect Camera

Trajectory



Example Frame



This trajectory primarily has vertical and horizontal movements. However the scene is quite deep, as you can see, behind the desk, but there is a significant number of close objects.

13.3 Result Collection

We now define the methods used determine the results for the algorithm evaluations.

13.3.1 Location Error

We collect this data as outlined in our previous discussion. However, we make one important change from the methodology used in the SLAMBench paper [16], that we align trajectory of the KFusion results. This is to make a fair comparison with LSD-SLAM which as we mentioned in Section 7.4, needs alignment.

13.3.2 FPS

We define the FPS as is the average time before a new frame can be processed in the process-every-frame mode.

KFusion In KFusion it is time of the entire sequential pipeline, i.e. computation only, no rendering.

LSD-SLAM In LSD-SLAM it is the time taken to be tracked and mapped (in the front end only) so that the if the key-frame changes the next frame is no tracked on an incorrect key-frame and therefore having to be discarded.

13.3.3 Energy Usage

SLAMBench originally only performed energy profiling on the ODROID (XU3) board. Between the publication of SLAMBench and the start of this project, power monitoring support was extended to the Intel platform, via their "Running Average Power Limit".

ODROID For the ODROID board energy measurements we use the sum of all the processors: Cortex-A7 and Cortex-A15 reported energy usages.

Seyward The RAPL feature of Intel processors, was designed with the aim of managing power usage. Energy usage, amongst other characteristics are provided via the Model Specific Registers, *MSR* [64]. For ease of use, we utilise the Performance Application Programming Interface, *PAPI* project, for access.
The Intel MSR registers provide three energy metrics: package, PP0 and PP1. The package measurement is of the whole CPU unit, where are PP0 is just the core components and PP1 is the non-core [64]. We use the package measurement.

Commentary

For these measurements we are relying on the manufactures to be accurate and also inclusive of all components. An alternative would be to measure the power of the whole computer, however this will include a lot of non necessary components and therefore it could skew the results.

13.4 Criticism's of this Methodology

One can argue this method is not very realistic. The primary argument is that characterising and optimising a SLAM algorithm within a framework and a set of fixed datasets could lead to skewed results, especially if the algorithm is tuned for the dataset and not the general case - i.e. some real-world interactions. This is a valid criticism, however, for the sake of repeatability, one has to use fixed datasets and operate experiments within a closed world. Moreover, given a good selection of datasets, which simulate the real-world, the behaviour within SLAMBench should be near real-world.

Chapter 14

KFusion Characterisation: Building Blocks and Kernels

We begin our analysis with KFusion, as the implementation of KinectFusion. We use this as the platform to later compare with LSD-SLAM, we therefore delay some analysis until our investigation into LSD-SLAM (Chapter 15). We follow the analysis from the SLAMBench paper but extend it where interesting comparisons can be made with LSD-SLAM.

14.1 SLAMBench Requirements

Unlike, our previous discussion with LSD-SLAM, it is trivial to support the SLAMBench requirements - these were already met when SLAMBench was published! By having this strictly sequential behaviour, means that enabling a process-every-frame mode is trivial, hence KFusion can be used as is.

The following sub sections describe our implementation choices and the parameters we are using to investigate KFusion.

14.1.1 Implementations

The SLAMBench authors ported KFusion to a variety of languages, targeting different platforms and architectures. These ports are shown in Table 14.1.

Since, LSD-SLAM is only utilises an x86 processor, in this section we only consider the C++ and OpenMP versions. Moreover, we actually only consider the OpenMP version as the C++ version is not optimised in any way, therefore leading to very biased results, namely the FPS will be very low.

Moreover, with respect to the current implementation, only one set of kernels, targeting a language / architecture can be used any any one time.

Implementation Name	Description
C++	Vanilla C++ implementation. Optimisations provided by the compiler.
OpenMP	The C++ version but with threading, using the OpenMP API.
OpenCL	A version utilising the OpenCL framework
CUDA	A version utilising the CUDA framework

Table 14.1: KFusion implementations available in SLAMBench.

14.1.2 Tunable Parameters

Along with selecting an implementation, there are a variety of parameters, so that the implementation can be turned for a particular scene, shown in Table 14.2. We use the default settings, except where noted.

We modify the default parameters due to the ICL-NUIM Office scene and the TUM RGB-D datasets being larger, we have to double the TSDF size (from $4.6m^3$ to $9.6m^3$) and resolution (from 256^3 to 512^3). However compared to the original SLAMBench paper each voxel represents the same volume: $0.01875m^3$.

The alternative to doing this is to minimise the size of the TSDF volume on a per dataset basis. However, this will mean that simply changing the dataset, will mean a possible change in a few parameters, which is not necessarily a fair test.

14.2 Building Blocks

We begin our characterisation of KFusion, with the building blocks. From Figure 14.1, we can see it is a simple sequential pipeline.

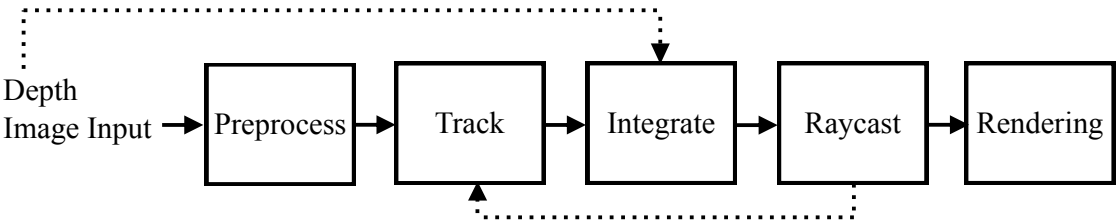


Figure 14.1: Building blocks of the KFusion implementation in SLAMBench.Dotted lines indicate data dependences, forwarded without computation [16].

If we extend Figure 14.1 by drawing the interactions amongst multiple iterations, shown in Figure 14.2, it is clear KFusion enforces a strict sequential behaviour due to to the data forwarding from the 'Raycast' to 'Tracking' stage. The 'Raycast' result must be available from the previous frame before the next frame can be tracked. Moreover, these two blocks are and the end and beginning of the pipeline, therefore there no building

Name	Description	Default	Program Argument
Compute Size Ratio	Scale input resolution.	2	`--compute-size-ratio`
Frames Per Second	The number of frames added to the system per second. This would override the process-every-frame mode.	0	`--fps`
ICP threshold	Minimum error for ICP to terminate	10^{-5}	`--icp-threshold`
mu	Minimum difference between calculated and expected value for a given TSDF cell update.	0.1	`--mu`
Initial Pose	(x, y, z) coordinates of starting position, within a unit cube	0.5,0.5,0	`--init-pose`
Integration Rate	Integrate on every n^{th} frame.	1	`--integration-rate`
Volume Size (m)	Dimensions of the volume	2,2,2	`--volume-size`
Tracking Rate	Skip tracking on every n^{th} frame.	1	`--tracking-rate`
Volume Resolution	The number of cubes each dimension is divided into	512,512,512[1]	`--volume-resolution`
Pyramid Levels	Number of levels in the pyramid for the ICP algorithm	10,5,4	`--pyramid-levels`
Rendering Rate	Frequency at which to render model, Hz.	4	`--rendering-rate`

Table 14.2: Parameters available to all KFusion implementations within SLAMBench.

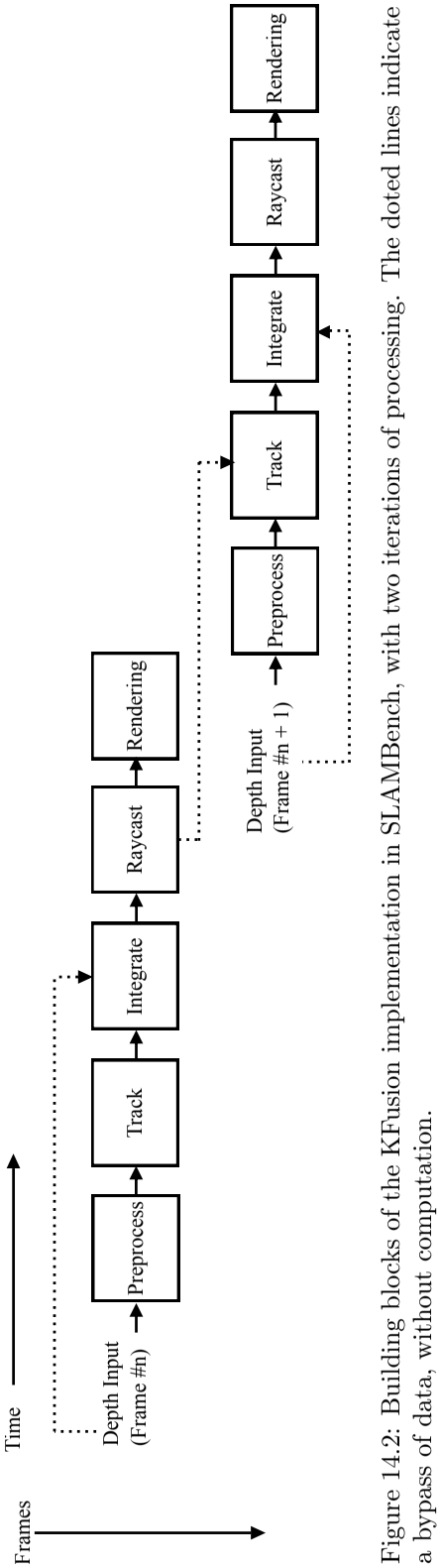


Figure 14.2: Building blocks of the KFusion implementation in SLAMBench, with two iterations of processing. The doted lines indicate a bypass of data, without computation.

blocks can be executed in parallel. (However, later we will see that inside processing blocks parallel techniques are used).

14.3 Kernels

KFusion can be further decomposed, from building blocks into kernels. This work was performed by the SLAMBench paper authors. We repeat these here for ease of reference in future sections. The 'In' / 'Out' columns refer to the input and output sizes to the kernel. (This is an amalgamation of Table 1 and Section IV Part B. from the SLAMBench Paper [16]).

Pre-process
The first step, prepares the depth image for the 'Tracking' and 'Integration' stages.

Kernel Name	In	Out	Description
mm to meters	Pointer	2D	Convert depth-map from meters to millimeters.
Bilateral filter	2D	2D	Apply filtering to reduce noise. (See Section 6.3.2).

Table 14.3: Pre-processing kernels in KFusion

Tracking
The 'Tracking' building block, primarily performs the ICP algorithm, as outlined in Section 6.3.3. This step calculates the transformation between the previous frame and the new frame.

Kernel Name	In	Out	Description
Half Sample	2D	2D	Sample the depth image to create a pyramid of down-sampled layers (a trick from Section 4.4).
Depth to vertex	2D	2D	Create a 3D point cloud for each layer in pyramid
Vertex to normal	2D	2D	Calculate a normal vector for each point in each layer.
Track	2D	2D	Find correspondence between the map/model and the point cloud created in the previous steps.
Reduce	2D	6x6	Sum up the differences between all corresponding points.
Solve	6x6	6x1	Solve a linear equation which results in the pose estimate update.

Table 14.4: Tracking kernels in KFusion

Integration
Given the pose transformation calculated in the tracking stage, the integration step merges in depth date, at the hopefully correct place into the TSDF (map for KinectFusion).

Kernel Name	In	Out	Description
Integrate	2D/3D	2D	Merge, 'Fuse', in the depth frame data, into the TDSF volume. (See Section 6.3.4)

Table 14.5: Integration kernels in KFusion

Raycasting
The tracking stage, when processing the next depth image, requires a view of the scene, with the latest depth information, therefore, after integration, the expected view of the scene is calculated by ray-casting.

Kernel Name	In	Out	Description
Raycast	2D/3D	2D	Capture a view of the TSDF volume at the current camera pose.

Table 14.6: Raycasting kernels in KFusion

Rendering
Separate from the main processing, but required for viewing the TSDF and any third party processing, such as feature extraction or debugging, will require a rendering of the TSDF.

Kernel Name	In	Out	Description
Render depth	2D	2D	Create a colour image of the depth map, where colour encodes depth.
Render track	2D	2D	Render an image of the tracking state for each pixel, e.g. successful, failure.
Render volume	3D	2D	Like the raycasting kernel, create a view of the scene at a particular pose.

Table 14.7: Rendering kernels in KFusion

14.4 Performance Investigation

Having described (the existing) decomposition of KFusion, we are in a place to begin to characterise it.

NOTE: All experiments have been run using the OpenMP variant of KFusion, except where noted.

14.4.1 Basic Performance Characterisation

We begin with performing an analysis of KFusion by varying the dataset and executing it on the two platforms - Seyward and ODROID. The SLAMBench paper only investigated the behaviour of KFusion with the ICL-NUIM Living Room Trajectory 2 dataset. We extend this here to many of the datasets mentioned previously.

The first performance analysis was performed using two ICL-NUIM datasets, Living Room trajectory 2 and Office trajectory 2, the findings of which are presented in Figure 14.3.

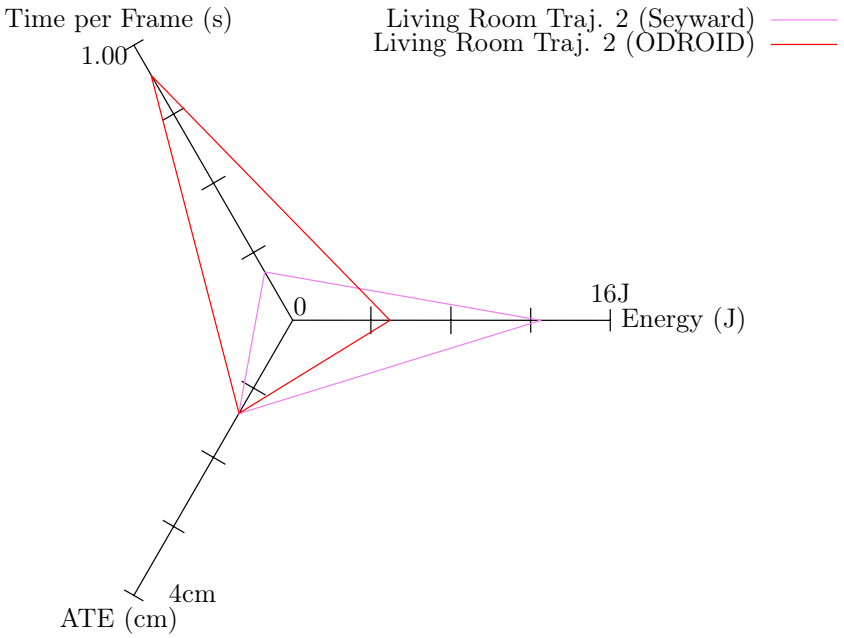


Figure 14.3: KFusion run with ICL-NUIM Living Room Traj. 2. Run on Seyward and ODROID using default parameters. Showing the three SLAMBench metrics.

As a sanity check, it is good to notice the ATE is consistent across platforms, with respect to a particular dataset. Furthermore, we can also get comparative results to that of the original SLAMBench paper. We compare with the un-aligned results, shown in Table 14.8. The slight change in ATE is caused by the change in volume size. We discuss at length the reasoning behind this, in Section 14.5.2.

Dataset	Original (Unaligned)	Our Results (Unaligned)
ICL-NUIM Living Room Traj. 2	2.07 cm	2.04 cm

Table 14.8: Comparison of results between SLAMBench Paper and our results (not aligned using the Horn method).

14.4.2 Extending Dataset Usage

Being able to reproduce the results, is good, but we can go much farther. We have experimented with the larger collection of datasets at our disposal. Figures 14.4, 14.5 and 14.6 shows some of these results.

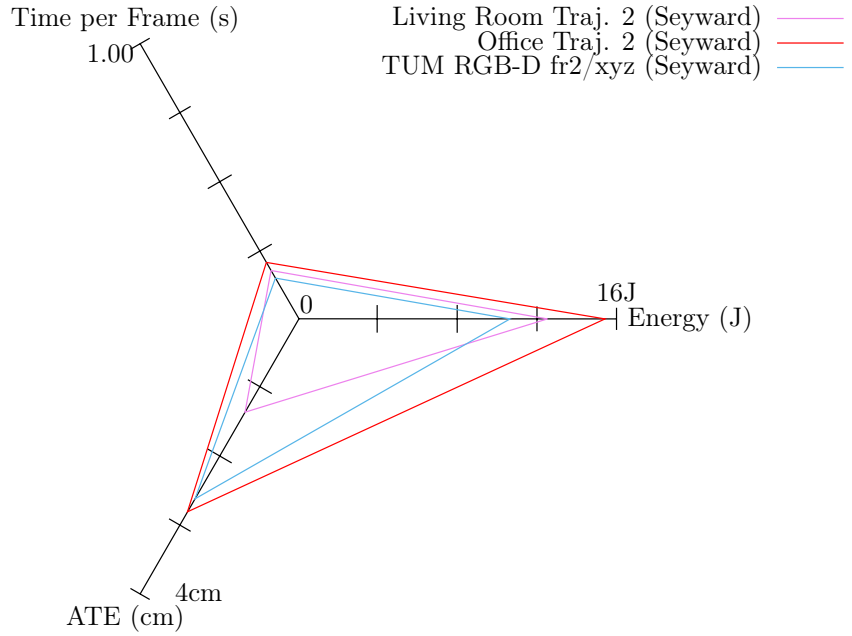


Figure 14.4: KFusion running under three datasets (ICL-NUIM Living Room Traj. 2, Office Traj. 2 and TUM RGB-D fr2/xyz), with default parameters running on Seyward, showing the three SLAMBench metrics.

We make a few preliminary observations, which we used to drive our future investigations:

- *Moderate Consistency* between energy and FPS, even after changing the dataset (see Section 14.5.1). And with a 'real' dataset, the TUM RGB-D fr2/xyz.
- *Poor FPS*. On Seyward, its about 4 FPS. On the ODROID it is about 1.1 FPS.

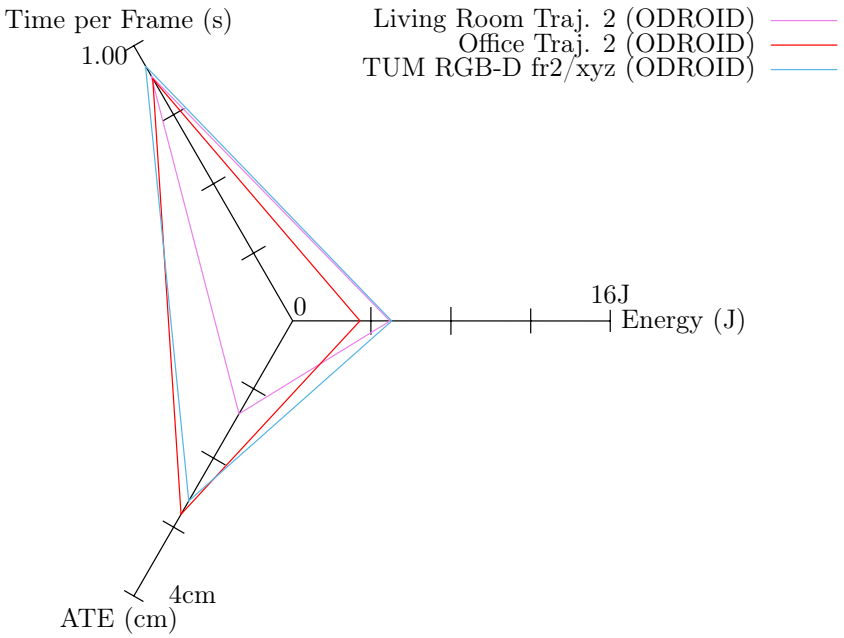


Figure 14.5: KFusion running under three datasets (ICL-NUIM Living Room Traj. 2, Office Traj. 2 and TUM RGB-D fr2/xyz), with default parameters running on ODROID, showing the three SLAMBench metrics.

- *Good ATE*, for the datasets, which generally follows a consistent percentage of the room size. For example, the ICL-NUIM Living Room trajectory 2 ATE is at about 0.8% of the room size (see Section 14.5.2).

But, what is seemly odd, is the excellent ATE for ICL-NUIM Living Room trajectory 1, in Figure 14.6. We investigate this in Section 14.5.2.

14.5 Characterisation for Three Metrics

So far, we have seen that KFusion is consistent, with respect to the three metrics, under the datasets we have tried. However, the parameters which have been used have been chosen, such that KFusion can track frames, but to understand their effect on the three metrics we need to perform further investigation. We combine our investigation of FPS and energy together (Section 14.5.1) and then focus on the ATE (Section 14.5.2).

14.5.1 FPS and Energy Dependencies

We have clearly demonstrated that the FPS and Energy are consistent for a given platform. We investigate the reasons for this; experimentally and analytically by inspecting both the sequential pipeline and some of the key kernels.

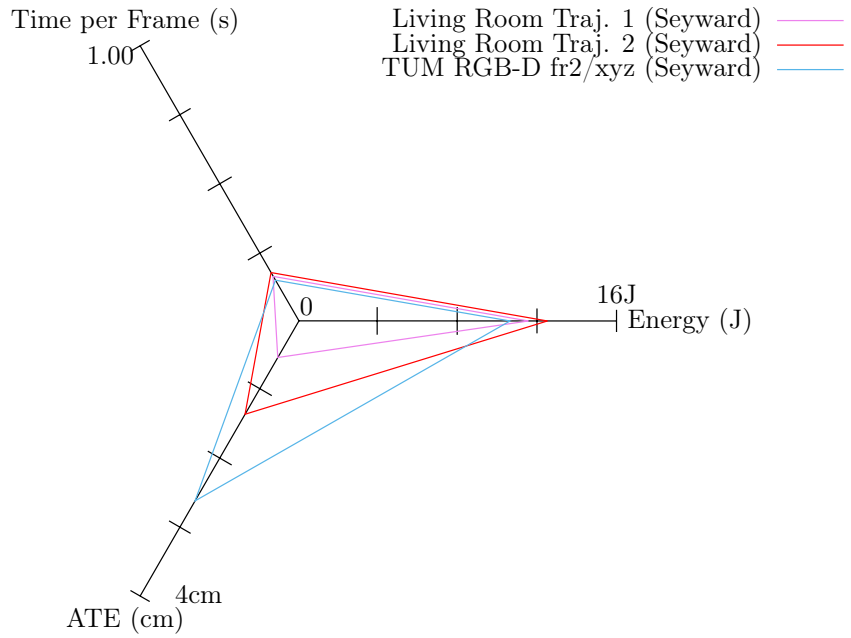


Figure 14.6: KFusion running under three datasets (ICL-NUIM Living Room Traj. 1, ICL-NUIM Living Room Traj. 2 and TUM RGB-D fr2/xyz), with default parameters running on Seyward, showing the three SLAMBench metrics.

In our first experiments, we saw repeatability across datasets, again we can see this when we plot (Figure 14.7) the timings of the kernels.

We can see the same repeating pattern that the 'Integration Kernel' is consuming the largest proportion of the total processing time. In an effort to understand the reasoning behind this, we varied the number of voxels within the TSDF volume, Figure 14.8 shows the results. (The same results are obtained for TUM RGB-D fr2/xyz are shown in Appendix A.1)

Clearly varying the number of voxels in the TSDF volume varies the amount of work. In Algorithm 2, we show the basic structure of the integrate kernel. This explains our observation as all voxels are accessed to check if a surface was detected within the voxel.

It is important to note that the three functions listed, are trivial and therefore do not incur any significant computation time. The reason that this has a proportionally large run time is caused by the sheer quantity of voxels to check, e.g. in the default setup is $512^3 \approx 134^6$ voxels.

Commentary

The 'Integration Kernel' and the number of voxels, combined, are clearly one of the largest contributing factors to the FPS and energy.

We can expand this analysis to all kernels within KFusion. Recall from Section 14.3,

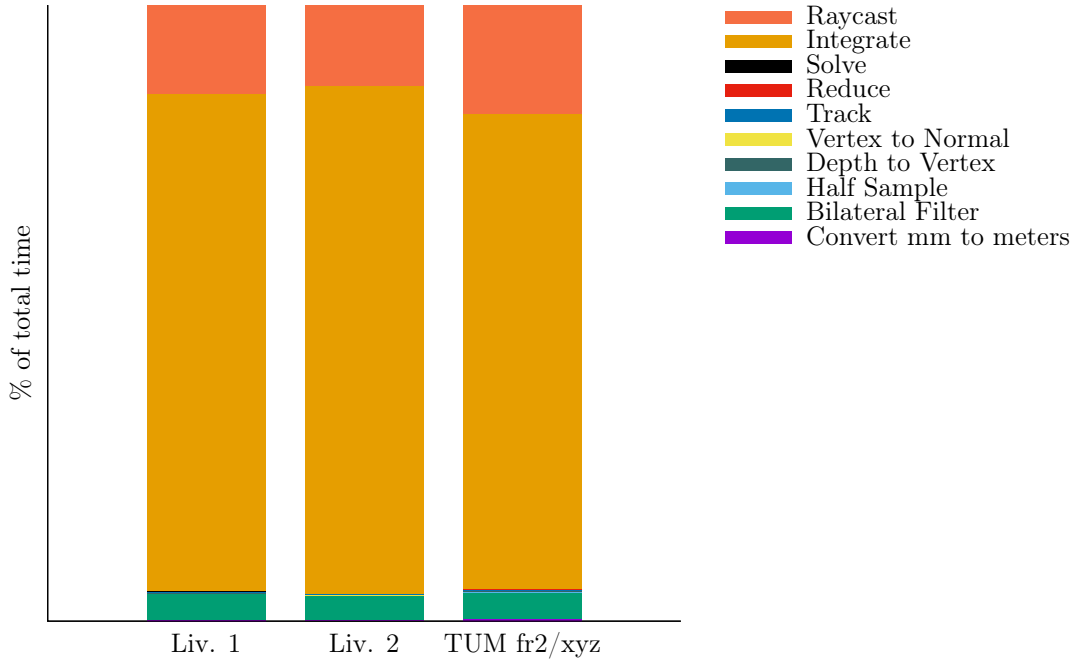


Figure 14.7: Kernel timings as a percentage of the total time, in KFusion running on Seyward. We vary the dataset (ICL-NUIM Living Room Traj. 2, Office Traj. 2 and TUM RGB-D fr2/xyz).

where we outlined the Kernels, as decomposed by the SLAMBench authors. All the kernels follow a similar process of iterating over all the pixels, and perform the same computation on all. The reason for highlighting this becomes clear when we investigate LSD-SLAM. (For those with different input/outputs, 'Reduce', 'Solve', this does not apply but they do not dominate the total time, so their behaviour is less interesting.)

Combining this consistent and predictable behaviour, with the sequential pipeline, therefore, given a set of parameters there is a fixed upper bound of work to perform. This results in a *predicable* frame throughput (inverse is FPS) and energy used per frame.

14.5.2 ATE Dependencies

We now turn out attention to the remaining metric, the ATE. In this section we aim to determine what parameters of KFusion and what features of the dataset affect the ATE.

To understand how the ATE is affected, lets work backwards from where the pose is determined, for each frame. The latest pose is calculated by aligning the depth frame with the expected view, using the ICP algorithm. This expected view was generated by raycasting from the TSDF volume using the previous frames pose. The TSDF volume was updated using the previously calculated pose and the previous depth frame. So, this shows there is a long dependency chain resulting from tracking, integration and ray-casting, which is tabulated in Table 14.9.

Algorithm 2 Integrate Kernel (High Level)

```
1: procedure INTEGRATE(volume, frame)
2:     for all y ∈ (0, volume.size.y] do
3:         for all x ∈ (0, volume.size.x] do
4:             for all z ∈ (0, volume.size.z] do
5:                 pos ← CALCULATEPOSITIONINFRAME(frame, x, y, z)
6:                 Δdepth ← CALCULATEDEPTHDIFFERENCE(volume, frame, pos)
7:                 if Δdepth > mu then                ▷ Previous estimate is way out
8:                     UPDATESURFACEPREDICTION(volume)
9:                 end if
10:            end for
11:        end for
12:    end for
13: end procedure
```

Component	Primary Dependent Properties
Tracking	• Raycast'ed view of the map
	• Number of pyramid levels and iterations per level.
	• ICP Threshold
Integration	• Number of voxels
	• Quality of map from previous iterations
	• μ
Ray Casting	• Map
	• Volume resolution
	• Pose (from tracking)

Table 14.9: Table of inter-dependences in KinectFusion / KFusion relating to the ATE calculation.

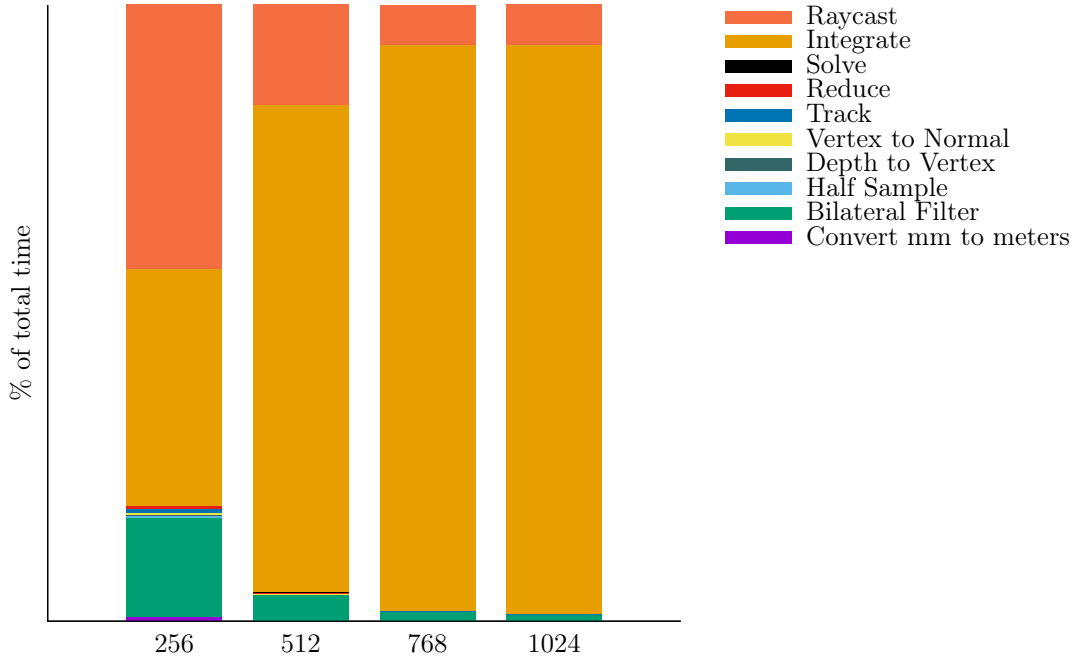


Figure 14.8: Kernel timings as a percentage of the total time, in KFusion running on Seyward. We vary the number of voxels (otherwise default parameters) on the ICL-NUIM Living Room Traj. 1 dataset.

Moreover, the noise in the depth measurements, will obviously affect all the building blocks, therefore the ATE.

Below we investigate the Tracking and Integration kernels as they exclusively affect the raycasting result.

Tracking

If we were only tracking, without updating the TSDF volume, the ICP threshold, will just affect the tracking. However, since KinectFusion / KFusion uses the tracking result to integrate the latest depth-map into the TSDF volume, any errors from the tracking will be integrated into the volume, possibly making the subsequent tracking of frames difficult and as a result tracking may fail. In Algorithm 3, we show the structure of the tracking building block.

There are two factors which can affect the ATE, even in a noise-less scenario: the number of iterations per pyramid level and the ICP threshold. Having too few iterations per pyramid level, will affect the result in the same way as having high ICP threshold, as the tracking will exit at a non-optimal result. Moreover, limiting the number of pyramid levels could mean the tracking gets stuck in a local minimum.

Interestingly, neither of these is the cause for ICL-NUIM Living Room trajectory 1 having such a good ATE. We compare, in Table 14.10, the type and quantity of exits

Algorithm 3 Tracking Building Block

```
1: procedure TRACKING(volume, frame)
2:     for all layer ∈ Coarse to Fine Layers do          ▷ Pyramid optimisation trick
3:         for 0 ≤ i < iterations for layer do
4:             trackingResult ← TRACKINGKERNEL()
5:             REDUCEKERNEL(trackingResult)
6:             poseUpdate ← SOLVE()
7:             pose ← pose × poseUpdate
8:             if ||poseUpdate|| < icp-threshold then
9:                 Break
10:            end if
11:        end for
12:    end for
13: end procedure
```

Dataset	**Number of Tracking 'Early' exits**	**Final** ||poseUpdate||
ICL-NUIM Liv. Room Traj. 1	98% (946 / 965 frames)	7.3×10^6
ICL-NUIM Liv. Room Traj. 2	97% (862 / 882 frames)	7.1×10^6

Table 14.10: Comparing the quantity and type of exists from the Tracking Building Block, using ICL-NUIM Living Room trajectories 1 and 2.

between the ICL-NUIM Living Room trajectory 1 and trajectory 2 datasets. We can see that they are comparable, and for the majority of the time they are exiting 'early'.

In order to ground this analysis, we have performed two experiments showing the effect of varying the ICP-threshold, on ICL-NUIM Living Room trajectories 1 and TUM RGB-D fr2/xyz, shown in Figures 14.9, 14.10. We can see that there is a small improvement in the ATE for the ICL-NUIM Living Room trajectory 1 dataset. However there is no change in the ATE of TUM RGB-D fr2/xyz dataset.

Integration

Turning over our attention to the integration step, recall that the volume is divided into voxels, which store a TSDF which describe distance to nearest surface. A smaller voxel, which discretises some space in the physical world, will enable a more precise, but not necessarily accurate, surface estimate, as smaller voxel, may encode more noise. We varied the number of voxels, on the ICL-NUIM Living Room trajectory 1 and TUM RGB-D fr2/xyz datasets, shown in Figures 14.11, 14.12. We can see that the dependence of the voxel resolution has the opposite behaviour across these two datasets. This highlights some complex interaction, but also that there *may* be a dependence on the dataset.

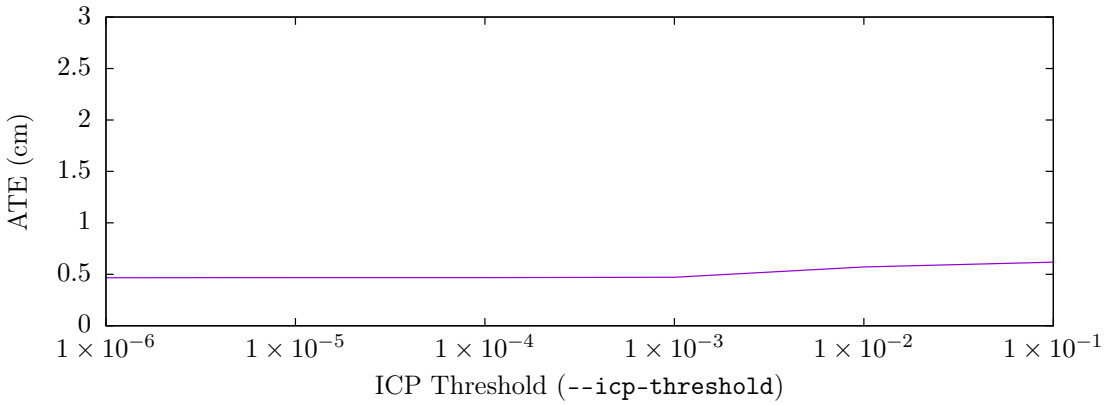


Figure 14.9: Varying the ICP-Threshold of KFusion using the ICL-NUIM Living Room Traj. 1 dataset

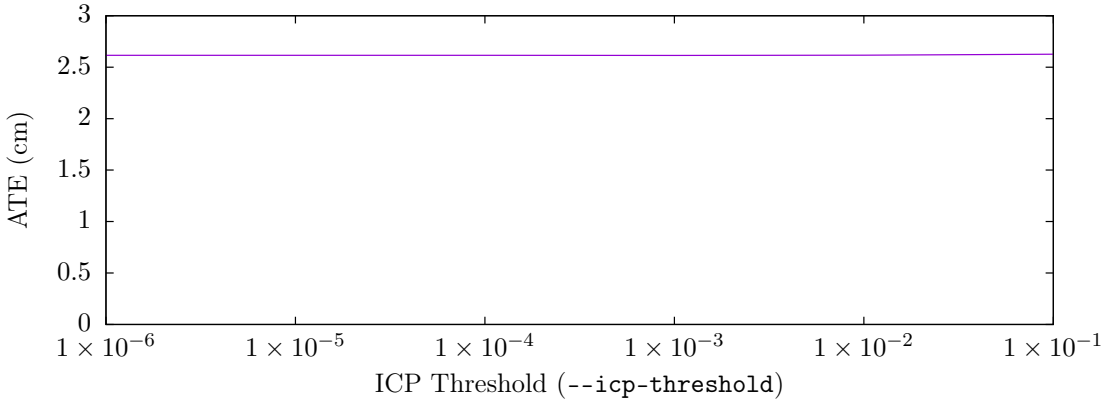


Figure 14.10: Varying the ICP-Threshold of KFusion using the TUM RGB-D fr2/xyz dataset

Conclusion

The minimum bound on the ATE is a complex interplay between the ICP threshold, number of voxels and voxel resolution, and mu, and also features of the dataset. We may not have used the optimal parameters, but what we have seen is consistency between the different datasets, with respect to room size. Though, we have seen an improvement of the ATE under the ICL-NUIM Living Room trajectory 1 and TUM RGB-D fr2/xyz dataset, this was caused by the small transformation between frames, playing directly into the KinectFusions small angle assumption.

14.6 Critical Commentary

The original KinectFusion algorithm, performs well for a limited set of scene configurations. However, some of the features which enable it to perform well, are also its downfall.

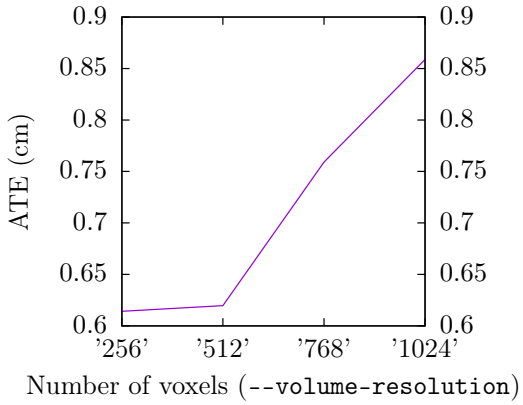


Figure 14.11: Varying the voxel resolution in KFusion under ICL-NUIM Living Room trajectory 1, with otherwise default parameters, on Seyward.

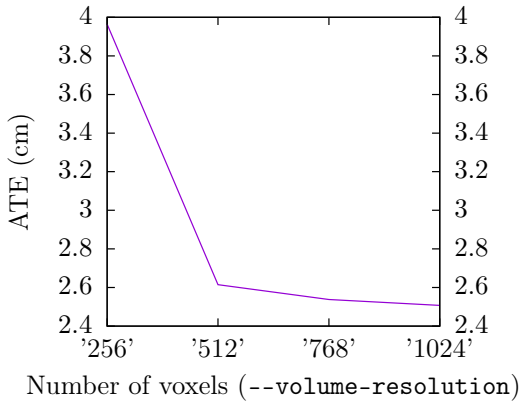


Figure 14.12: Varying the voxel resolution in KFusion under TUM RGB-D fr2/xyz, with otherwise default parameters, on Seyward.

We investigate each of these in turn.

14.6.1 Truncated Signed Distance Function (TSDF)

The choice of data structure (TSDF) enables a dense, detailed model to be generated (the precision is user configurable) however, the model size is limited by the available memory on the host platform, hence a trade-off between precision and scene size.

An extension to solve the limited scene size was devised by Whelan *et al* which was realised in 'Kintinuous'. It allows the model to be dynamically resized, so the precision can remain constant but the size of the scene need not be known at runtime [65].

Furthermore, although the data structure is dense, the interesting data - that is the zero crossings, the object edges - is very sparsely distributed through the data structure, therefore there is a lot of redundant data. There have been various methods to compress

the TSDF volume. One published method by Zeng *et al* [66] uses an octree, which according to the authors reduced memory usage by 10%. Another approach is to perform Voxel Hashing [67], so that only voxels with data are stored.

14.6.2 Input Device

While the Microsoft Kinect Camera is a commodity item, thus making the algorithm more accessible, it is, however, limited in functionality.

Firstly, the depth camera. It can only determine depths between 800mm and 4000mm [68], therefore if the scene is smaller or larger than this the model will potentially be inaccurate. So, this works well with small indoor scenes with limited depth. Moreover, it determines depth by projecting a known image onto the scene, and by seeing how it is deformed to generate a depth map. This is performed within the infra-red, *IR*, portion of the electromagnetic spectrum [69]. Clearly, the depth map will be an approximate, and also it will be susceptible to IR noise. Microsoft recommend, for using a Kinect with an XBox - clearly applicable here - that the surfaces are well lit, but not in direct sunlight, as this will introduce too much IR interference [70].

Furthermore, KinectFusion only utilises the depth data and not the RGB data, therefore potentially useful data is being discarded.

14.6.3 Parallel Processing Architecture Requirement

The original KinectFusion implementation utilised a GPGPU as the core processing platform. The SLAMBench authors experimented with porting KFusion to different parallel and sequential architectures, the results of which were shown in Figure 4 in their report, reproduced here in Figure 14.13.

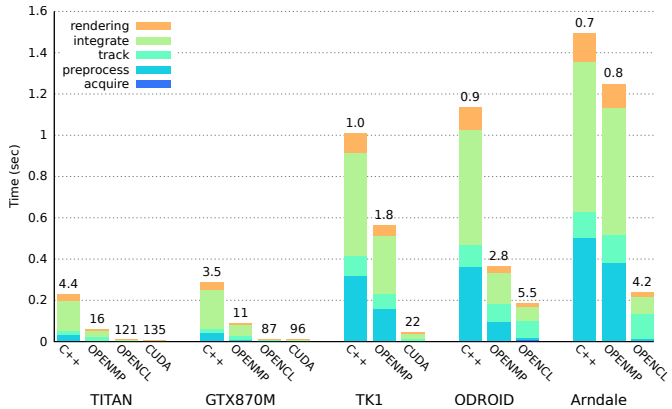


Figure 14.13: KFusion building blocks, with FPS on top of histogram. 'Titan' is equivalent to Seyward but with a powerful GPU (NVIDIA TITAN). The ODORID is identical to the one we used. The TK1 and Arndale are other embedded platforms. GTX870M is a laptop with an NVidia graphics card of the same name. Reproduced here from Fig 4 in SLAMBench paper [16].

Clearing from the diagram only the implementations utilising a parallel architecture and associated language, such as OpenCL and CUDA have a suitable FPS, which makes their use required for a good level of performance.

KinectFusion utilises the GPU in order to speed this process up, as each voxel - could be 512^3 voxels - in the TDSF volume needs to inspected and possibly updated.

Since one thread cannot be launched for each voxel, a single thread needs to interact with numerous voxels. KinectFusion makes an optimisation by coalescing the memory accesses. Each thread operates on a slice of the volume. A slice operates over the z axis, with the x, y being fixed. This means memory access is sequential, as the threads operate in lock step (on the GPU), thus allowing memory accesses to be combined, or *coalesced* [9].

14.7 Summary

We briefly outline some of the interesting features we have discovered from our work here, which we will carry forward into our comparison with LSD-SLAM.

We have seen that KFusion is has a consistent worst case, which has a reasonable ATE, and predictable energy usage per frame and (but poor) FPS. We have seen how the integration - the map updating - dominates the processing time.

We investigated the functions affecting the FPS, energy and ATE. We showed how KFusion is predicable with respect to FPS and energy, due to the fixed upper bound of work - a by product of a dense algorithm. We also highlighted that the computation time is dominated by the 'Integration Kernel'. With regards to the ATE, we showed that there is a complex interplay between the parameters of KinectFusion and the dataset. However, with the datasets we have tried, up to this point, the results are fairly consistant as a percentage of the room-size. (However, this does not always hold as we will come to see in the next chapter.)

Chapter 15

LSD-SLAM Characterisation: Building Blocks and Kernels

We now turn our attention to the LSD-SLAM algorithm and the selected implementation. We follow a similar style of characterisation and evaluation to our work with KinectFusion / KFusion, except we expand our analysis across four chapters. Firstly, in this chapter, in order to gain an understanding how this implementation operates we perform a three step characterisation, firstly as a single unit then decomposing it into building blocks and finally into kernels (Chapter 16). In the subsequent two chapters, we explore the effect of parameter values (both software and hardware) as a design space exploration (Chapter 17) and finally, investigate how LSD-SLAM operates in the 'real world' (Chapter 18).

15.1 Basic Performance Characterisation

To begin, we run LSD-SLAM with a selection of the datasets, defined in Section 13.2.3, to gain a basic understanding of how it performs. This is shown in Figures 15.1, 15.2.

15.1.1 Sanity Checking

Again, this provides a sanity check to test our integration, by comparing with the original paper. Importantly, the original LSD-SLAM paper uses the RMSE for the ATE, we we are using the MAE, hence we need to compute and compare the RMSE values. The comparison is presented in Table 15.2.

Dataset	**Original (RMSE)**	**Our Result (RMSE)**
TUM RGB-D fr2/xyz	1.47 cm(RMSE)	1.46 cm(RMSE)
TUM RGB-D fr2/desk	4.52 cm(RMSE)	2.67 cm(RMSE)

Table 15.1: Comparison between the LSD-SLAM paper [15] and our results. (Using default parameters, run on Seyward.)

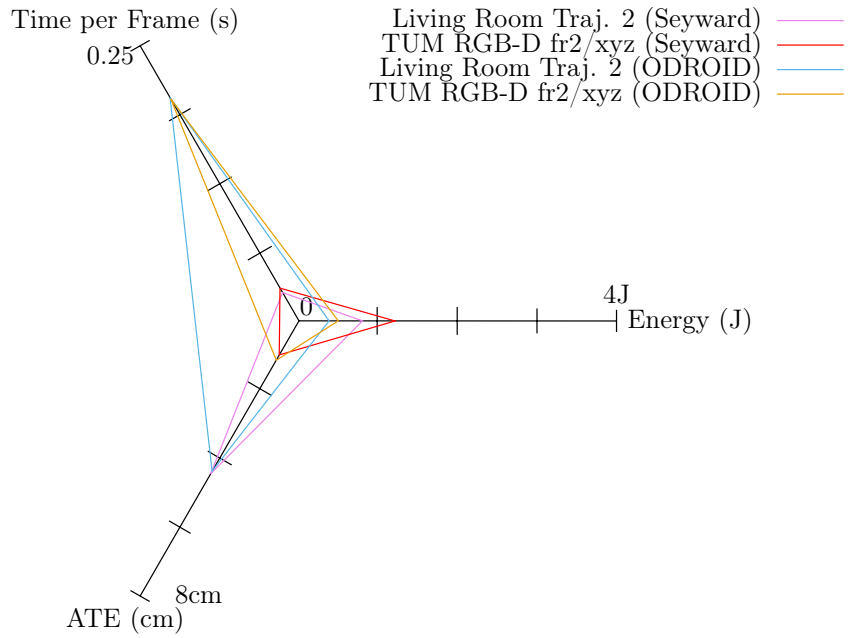


Figure 15.1: LSD-SLAM operating on a variety of datasets using default parameters, on both Seyward and ODROID.

As you can see, we can recreate the results for the TUM RGB-D fr2/xyz dataset but the performance of the TUM RGB-D fr2/desk dataset is better. We argue that this is acceptable for the following reasons: firstly, in our correspondence with Jakob Engel (an LSD-SLAM author) he notes that the version of code we are using and the version used for generating the results in the paper are different; secondly, it is better so the differences between the versions must be improvements - over time, one would not want to make their algorithms worse.

15.1.2 Comparing with KFusion

We can see from comparing with KFusion, LSD-SLAM *can* perform substantially better, in all metrics.

- **Frame throughput and Energy**
  - On the desktop, Seyward, LSD-SLAM achieves a super real-time frame processing rate.
  - Like KFusion, LSD-SLAM cannot perform in real-time on the ODROID.
- **Accuracy**

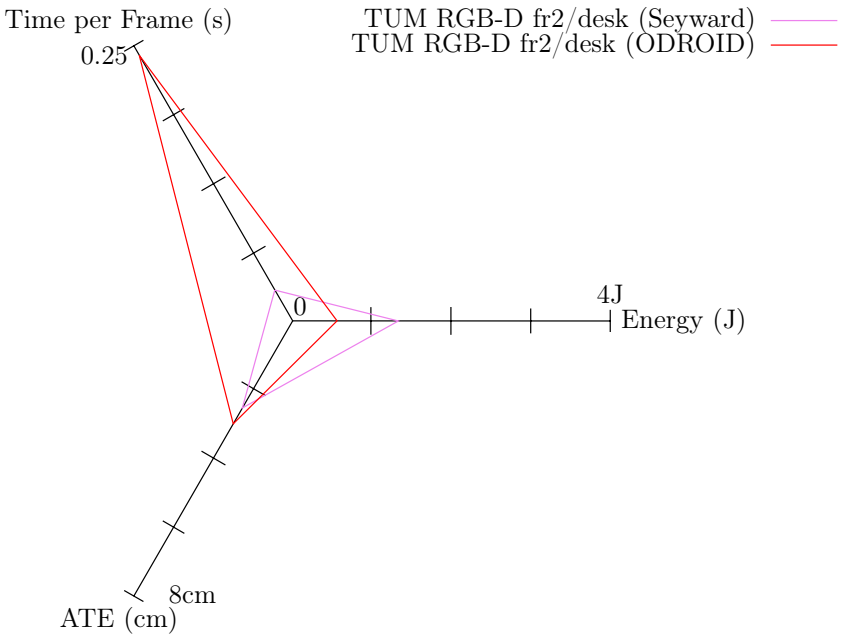


Figure 15.2: LSD-SLAM operating on the TUM RGB-D fr2/desk dataset using default parameters, on both Seyward and ODROID.

- Accuracy is far more varied. It is not a percentage of a the room size like KFusioon.
- It completely fails to track under the ICL-NUIM Office Scenes and Living Room trajectory 1.

In the following sections we investigate the last two observations.

15.1.3 Tracking Failure with Some ICL-NUIM Scenes

There is a complete failure of tracking with regards to the Office Scene. A probable cause of this is the combined trajectory and the reasonably texture-less scene[1].

In the Office 0 and 2 trajectories the camera spends significant proportion of the time looking at the ceiling and walls, which are fairly texture-less as well as the minimal change in light intensity therefore there are no shadows or other lighting artefacts. This means there is there a small number of suitable gradients to track, which cause the tracking loss.

We delve deeper into this, when we investigate the 'Minimum Gradient Threshold' parameter, in which we show how tracking failure can be 'achieved' under the Living Room trajectory 2 dataset (in Section 17.2.1).

[1]We recommend the reader to view the videos, composed of the frame from the dataset, available here: `http://www.doc.ic.ac.uk/~ahanda/VaFRIC/iclnuim.html`

15.1.4 Spread of Trajectory Errors

LSD-SLAM appears to perform, for the ATE metric, much better than KFusion, for a subset of the tracked scenes. However, by only comparing with the MAE we hide differences, specifically the spread of the absolute trajectory errors. This provides interesting insight into the differences between the algorithms. In Figures 15.3, 15.4 we plot, using a histogram, the distribution of ATE's obtained, under the ICL-NUIM Living Room Trajectory 2 dataset.

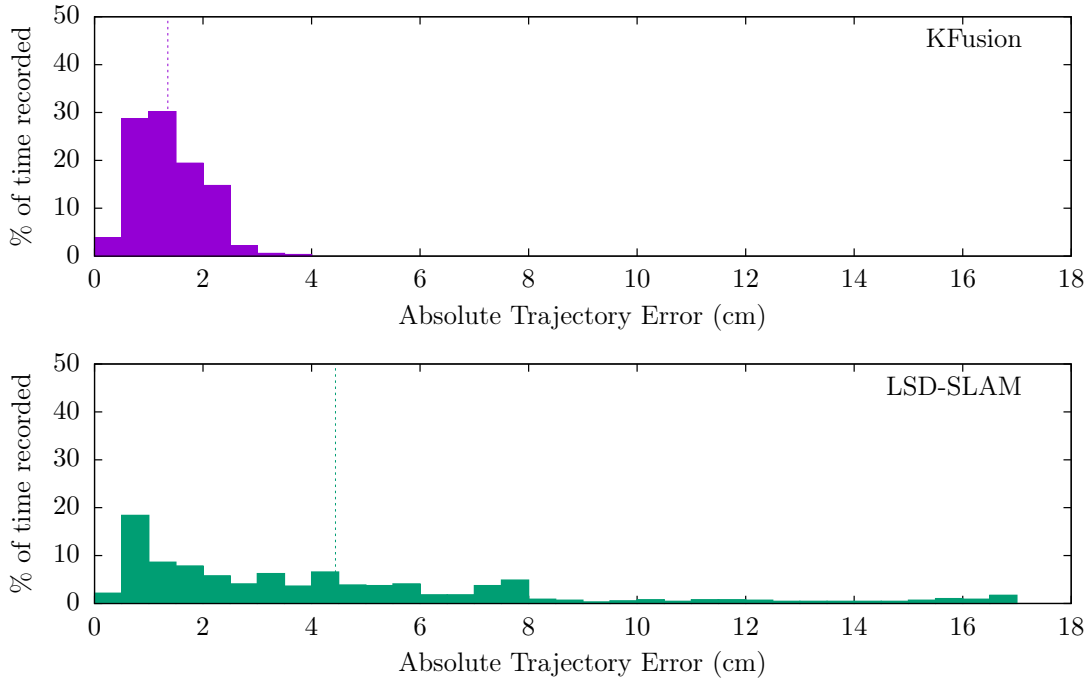


Figure 15.3: Distribution of ATEs under the ICL-NUIM Living Room Traj. 2 Dataset, using KFusion and LSD-SLAM. The MAE's is highlighted. (Default parameters and run on Seyward)

We can see that KFusion performs strictly better (with respect to ATE) than LSD-SLAM, from Figure 15.3. However, for the 'real world' dataset, TUM RGB-D fr2/xyz, LSD-SLAM performs better (Figure 15.4), but they both have a long-tail of errors. But the MAE's are fairly similar, KFusion's is worse by 1cm, but LSD-SLAM it has a much larger proportion of smaller errors. This highlights using RMSE is better, as it encodes some of the spread, "[the MAE] gives less influence to outliers" [23]. We compare the RMSE and MAE in Table 15.4, for the TUM RGB-D fr2/xyz.

However, it is also worth noting the the spread of errors for the ICL-NUIM Living Room Trajectory 1 dataset under KFusion, as it preforms very well. We show this in Figure 15.5, along with all the other trajectories from the ICL-NUIM Living Room scene.

If we consider all these results together, from Figure 15.3 to 15.6, we can make the further following observations.

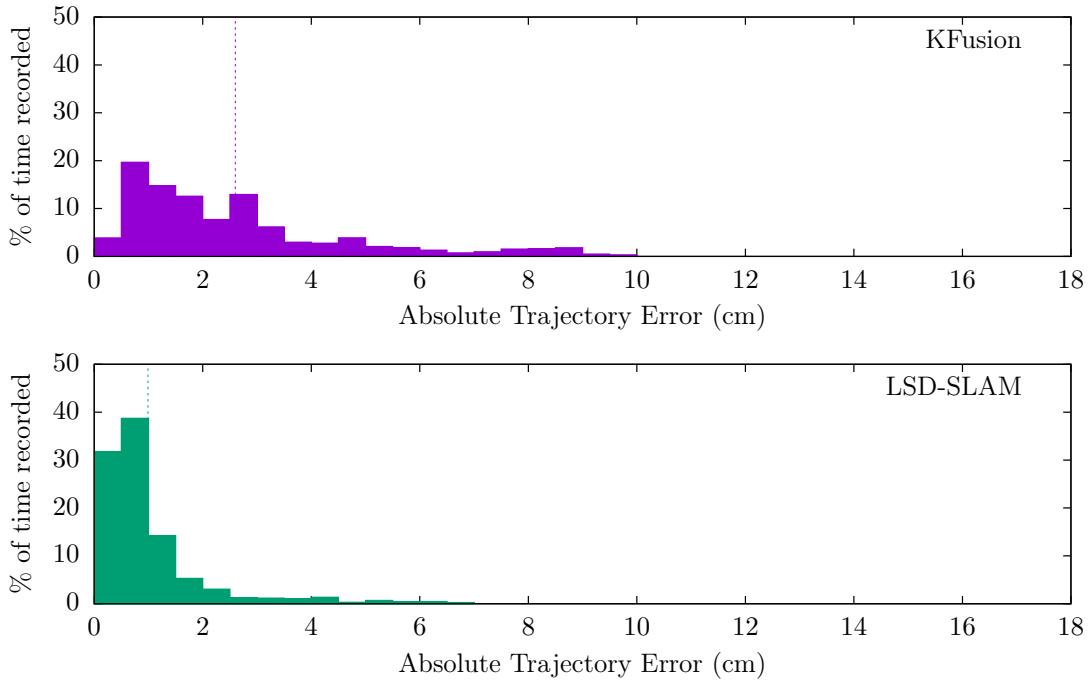


Figure 15.4: Variation of ATE under the TUM RGB-D fr2/xyz Dataset, using KFusion and LSD-SLAM. The MAE is highlighted. (Default parameters and run on Seyward)

Algorithm	**RMSE**	**MAE**
KFusion	3.35 cm	2.62 cm
LSD-SLAM	1.46 cm	0.99 cm

Table 15.2: Comparing the use of RMSE and MAE as ATE metrics for evaluating a SLAM algorithm. Values from default parameters using TUM RGB-D fr2/xyz.

- These algorithms show a high dependence on the dataset. LSD-SLAM is particularity susceptible to poor texturing from the synthetic dataset, where as KFusion appears to be more robust.

- The algorithms either perform well, in which case their is minimal variation (particularly in KFusions case with ICL-NUIM Living Room trajectory 1). However, when they perform poorly, both the mean and spread of errors increase. Particularly in LSD-SLAM's case tracking can fail entirely.

These two factors mean that for any evaluations of algorithms, the range of datasets must be large so there is a reduced chance of bias. Moreover, it highlights the problems of using synthetic datasets, especially when the sparse method relies on good lighting and texturing.

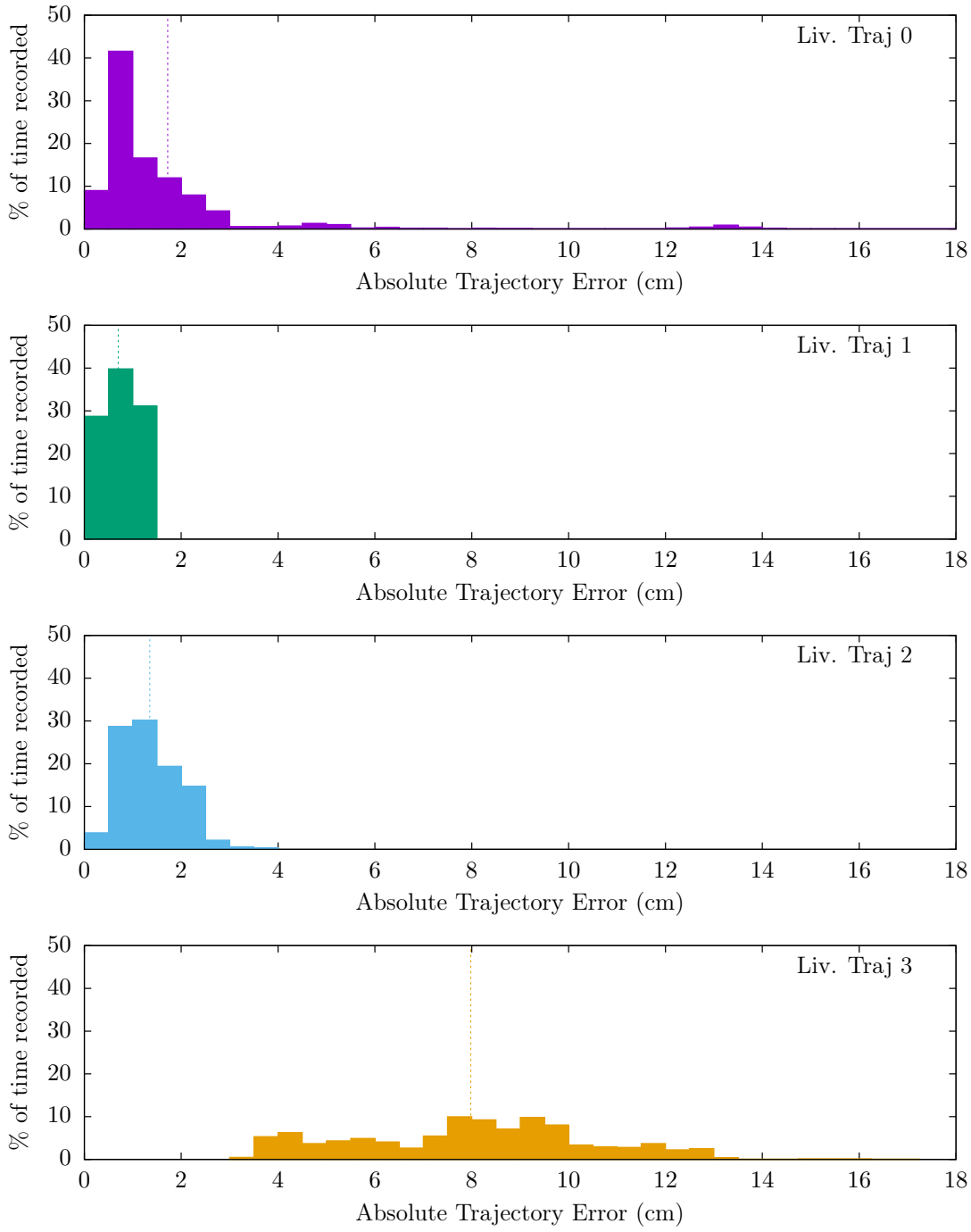


Figure 15.5: Variation of the ATE's under all trajectories in the ICL-NUIM Living Room scene, as run under KFusion. The MAE's are highlighted. (Default parameters and run on Seyward)

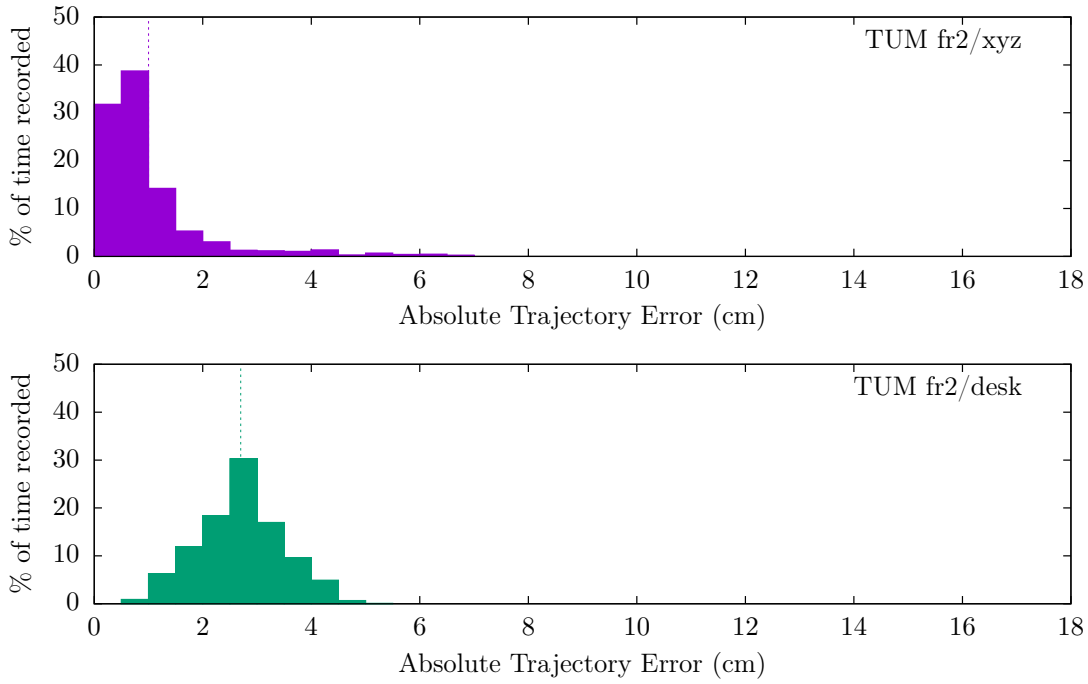


Figure 15.6: Variation of ATE under the TUM RGB-D fr2/xyz and fr2/desk datasets using LSD-SLAM. The MAE is highlighted. (Default parameters and run on Seyward)

15.1.5 Summary

We can see that the variation of trajectory errors varies substantially depending on the dataset used. For datasets which 'play into the algorithm', there is a small trajectory error. But, we have seen that LSD-SLAM is very susceptible to lack of texturing in the synthetic ICL-NUIM dataset. Furthermore, for those scenes which can be tracked by LSD-SLAM it performs better in all three metrics: a good FPS (greater than real-time), ATE and energy per frame usage.

However, to substantiate the claims about LSD-SLAM performing better than KFusion, (highlighting semi-dense it better than dense) we need to perform an exhaustive DSE to fully state these claims. We also need to fully optimise KFusion to be able to compare like-for-like.

This variation must be taken into consideration when considering integrating LSD-SLAM into motion planning algorithms, such as quadcopters, since at some points in time the estimated location could be off by upwards of 8-10cm. (There are practical methods such as filtering which can help combat this sort of variation.)

15.2 Building Blocks

We can develop a further understanding of LSD-LSAM if we decompose it into building blocks. We, again, treat the threads as building blocks as they each have a specific, discrete role.

NOTE: Throughout the following sections, where we have performance measurements, we have consistently used TUM RGB-D fr2/desk dataset run on Seyward, except where noted.

15.2.1 Tracking and Depth Mapping

We begin with the first two threads, namely 'Tracking' and 'Depth Mapping'. Recall, as part of the work to provide a process-every-frame mode these two threads are 'tied' together. To begin to understand their behaviour, we need to consider which paths are taken most frequently, so that we can attempt to determine the critical path. This is shown in Figure 15.7.

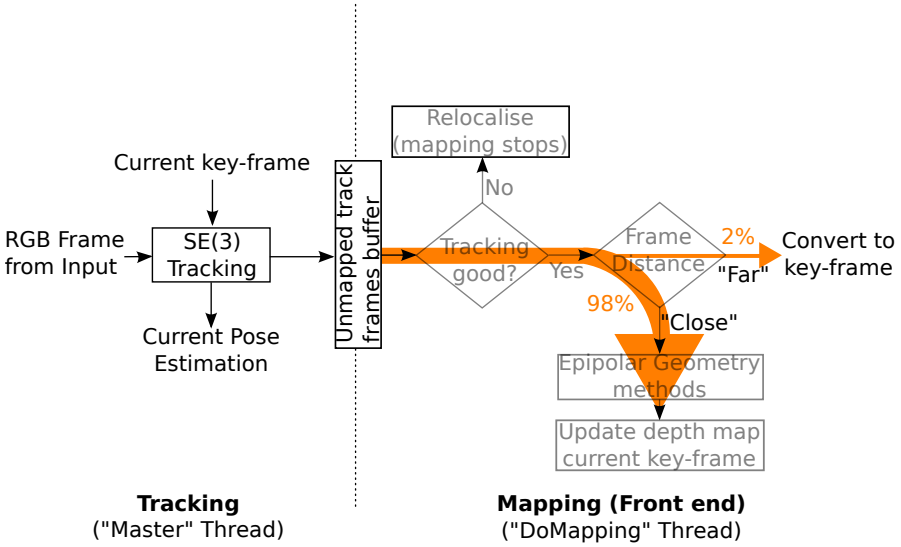


Figure 15.7: Ratio of frames updating a depth map vs being promoted to key-frame status. (Again, tracking failure and re-localisation is ignored.)

Using the ratio of paths taken, which is clearly biased to the updating of the map, we can generate a pipeline view, (like we did for KFusion in Figure 14.2), for these two threads. This is shown in Figure 15.8

We now compare with KFusion with respect to the pipeline structure and map update method, separately.

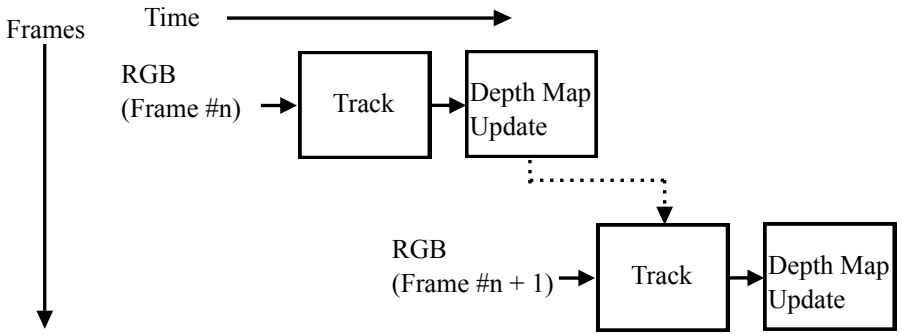


Figure 15.8: Pipeline of LSD-SLAM Tracking and Mapping Stages (taken 98% of the time). Dotted line shows data-dependency of the key-frame's depth map.

Pipeline Structure

Comparing with the KFusion pipeline (shown in Figure 14.2), there is a similarity in that they both have a dependency between iterations, namely their 'map'. LSD-SLAM algorithm's dependency structure is such that only two stages are needed before the depth-map can be used in the next iteration, unlike KFusion, which requires processing to have completed in all of the stages.

These two stages, **SE**(3) tracking and the depth-map update take on average 23 ms, each accounting for half of the time. (This is not an entirely fair comparison as only the tracking stage has been optimise to use SIMD instructions (SSE on X86 and NEON on ARM)).

The other path, taken 2% of the time, promotes the current frame to a key-frame if it is determined to be too far away. This block takes 17ms, therefore we consider the depth update path to be the *critical* and 'common' path.

An important similarity of these two algorithms is their dependence between iterations. They both update a representation of the scene (KFusion: TSDF Volume; LSD-SLAM: depth-map), and use that for tracking the next frame. Moreover, for tracking purposes they both rely on the assumption that the map is perfect, then reversing the assumption, by assuming perfect tracking to be able to update their respective maps.

Map Update

Exploring the similarity between the two algorithms further, the map updating stages are both a source of their bottle-necks. LSD-SLAM on each iteration updates a 2D depth-map, whose size is the same as the input resolution[2].

Given input resolution :	$width \times height$
The depth-map update is:	$O(width \times height)$

[2]The depth-map size can be changed, as was done to get LSD-SLAM operational on a mobile platform [14], but we keep them the same for our experiments.

An issue with describing the update using the asymptotic behaviour is the update always operates using far fewer pixels the full resolution[3], as LSD-SLAM is semi-dense. Furthermore, the method of selecting pixels for the purposes of depth estimation (the mapping phase), can be controlled (via the minimum gradient threshold) and is also dependent on the scene structure and lighting conditions. We investigate this in our design space exploration, Section 17.2.1.

KFusion is unlike LSD-SLAM in its map update procedure, 'Integration'.

Given TSDF Volume Size:	$VolumeWidth \times VolumeHeight \times VolumeDepth$
The TSDF update is:	$O(VolumeWidth \times VolumeHeight \times VolumeDepth)$

The total work to perform here is fixed, at initialisation time. But as you can see this is much larger and for any reasonable size, e.g. 128, 256 or larger, the update size is substantially bigger.

Number of Voxels	**Mean Integration Time (s)**	**Per Voxel Time (s)**
256^3	0.0093	5.5478×10^{-10}
512^3 (Default)	0.0739	5.5029×10^{-10}
1024^3	0.5120	4.7685×10^{-10}

Table 15.3: Integration times of KFusion using default parameters under ICL-NUIM Living Room 2.

Recall from our previous discussion, that KFusion is consistent, these times do not vary much when changing datasets.

Promotion to Key-Frame

This sub-building block is not interesting at this level, much more can be said when we delve into the kernels.

15.2.2 Constraint Finding and Optimisation

We now turn our attention to the second half of LSD-SLAM at the building blocks level, which is composed of 'Constraint Finding' and 'Optimisation'.

LSD-SLAM is still a SLAM system without these two components however it cannot detect loop-closures, and therefore misses out on the potential to improve the pose estimates. We have already discussed the behaviour of these blocks in Chapter 10 when we were creating the process every frame mode, but we have not seen the effect of disabling this feature. In Figure 15.9, we show the behaviour of LSD-SLAM with and without these two components.

[3]The implicit multiplier c is always: $c \ll 1$.

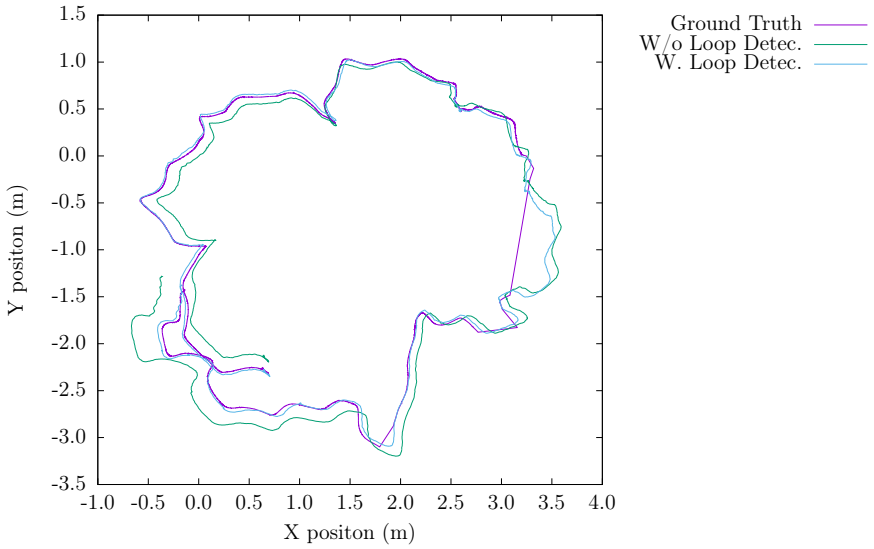


Figure 15.9: LSD-SLAM using TUM fr2/desk to highlight difference in the calculated trajectory between enabling and disabling loop-detection and pose optimisation. (Run on Seyward with default parameters.)

Even in this small scene, with one loop-closure, the pose optimisation does improve the calculated trajectory. However, this would be expected to have a much larger effect on large scenes with many loop closures.

15.3 Summary

We have studied LSD-SLAM in three levels: a complete unit, as a set of building blocks and at the kernel level. This has enabled us to see how LSD-SLAM can perform, especially when compared with KFusion.

Taking LSD-SLAM as a complete unit, we have seen with respect to the three metrics:

- Sometimes better performance when it *can* track.
- Tracking failure for many ICL-NUIM Datasets.
- In particular the spread of errors, where we noted that LSD-SLAM can perform significantly better than KFusion.

When we decomposed it into building blocks, and we showed the critical and common path was to tracking the frame, then update the depth map (taken 98% of the time). Comparing the pipelines of building blocks, the time before a new frame can be tracked is much shorter in LSD-SLAM, than KFusion, as the map updating is cheaper. This emphasised the how the parallel, asynchronous pipeline can lead to faster time between tracking due to the shorter update time.

Throughout this section, we have noted how the the sparse filtering of intensity gradients in LSD-SLAM makes its subject to the dataset especially if there is not suitable lighting and textures.

Our primary contributions, to the field, from this chapter are:

1. Dependence on the dataset
    - KFusion is more robust than LSD-SLAM for tracking on the datasets we have used.
    - Especially on 'real world' datasets (TUM RGB-D), LSD-SLAM can maintain tracking, and perform in real time.
2. Need for a full design space exploration, for result validation.

Chapter 16

Kernels by Building Block

We continue our investigation by further decomposing LSD-SLAM into even smaller components. In this final decomposition, we decompose the building blocks into kernels. The purpose of decomposing LSD-SLAM into building blocks is to[1]:

1. Investigate how the implementation is constructed
2. Identify 'hotspots'

There are many kernels within LSD-SLAM, therefore we focus on each building block individually.

16.1 Process / Master Thread

The master thread collects frames from a source, which could be an RGB camera or in SLAMBench's case a RAW file generated from either a TUM or ICL-NUIM dataset. These frames are then processed by LSD-SLAM, the first part of which is implemented and executed within this thread. The algorithm for this step is outlined in Algorithm 4.

16.1.1 SE(3) Tracking

Central to the tracking stage, is the **SE**(3) tracking method, both in terms of processing time and functionality. This processing block performs the algorithmic step outlined in Section 7.3.2, where incoming frames are tracked against the current key-frame's depth map. The algorithm, as provided in the implementation (in the `SE3Tracker` class), is outlined in Algorithm 5.

This procedure starts at a coarse layer (using the pyramid trick from Section 4.4), and at each layer (attempts to) improve the estimate of the transformation, ξ, between the new frame and the key-frame, by minimising the photometric error. As we have previously mentioned this procedure utilises the Levenberg–Marquardt algorithm. It can

[1]This also provides the basis for future work, when extracting the kernels to investigate alternatives, in a mix-and-match approach.

Algorithm 4 Master Thread

```
1: procedure MasterThread()
2:     for all frame ∈ Frames do
3:         trackFrame(frame)
4:     end for
5: end procedure

6: procedure TrackFrame(frame)                ▷ Track a frame using SE3 tracking
7:     transformation ← SE3Tracker(key-frame, frame)
8:     if transformation.translation() > MAX_DISTANCE then
9:         createNewKeyFrame ← true           ▷ Changed in Depth-Est. Build. Block.
10:    end if
11:    unmappedTrackedFrames.append(frame)
12: end procedure
```

been seen (lines 11-22 in Algorithm 5) that the heuristic to determine the λ starts at some guess $\lambda_{initial}$ calculating the error then increases λ by multiples of v, again evaluating the error function until a descent direction is found[2].

16.1.2 Re-localisation

This step can fail (not shown here in Algorithm 4), and if it were to do so re-localising is performed. Re-localisation attempts to find the best matching key-frame from the pose graph for the frame. In the current LSD-SLAM implementation, once re-localisation starts, the map is invalidated and frames are directly added to the re-localiser and no more processing takes place.

16.1.3 Kernels

As you will notice in Algorithm 5, the calculation is broken down into a series of steps: `CalculateResidual`, `CalculateWeights`, and `CalculateJacobian`. Although these could be combined into a single kernel, the inner loop of the Levenberg–Marquardt Algorithm (lines 11-22) does not need the Jacobian to be recalculated, hence there is no call to `CalculateJacobian`. Moreover, not shown in Algorithm 5, is that there are checks performed on some results so that the procedure can be abandoned early if the frames are not suitable for some reason e.g. there are not similar enough.

Therefore, clearly the algorithm can be decomposed into the following kernels, shown in Table 16.1:

[2]The outline of the **SE**(3) tracking algorithm (Algorithm 5), as from the code, was understood through a combination of Algorithm 1 in 'Odometry from RGB-D Cameras for Autonomous Quadrocopters' [60] and original discussions between Emanuele Vespa and myself

Algorithm 5 SE(3) Tracking Methods

procedure SE3Tracker(key-frame, frame, ξ_{initial})
  $\xi \leftarrow \xi_{initial}$ ▷ Uses ξ from last frame

  for all layer ∈ Coarse to Fine Layers **do** ▷ Pyramid optimisation trick
    $\lambda \leftarrow \lambda_{initial}$ ▷ $\lambda_{initial}$ is user definable
    $i \leftarrow 0$
    for $i <$ layer.maxIterationCount **do**
      $r \leftarrow$ CalculateResidual(layer)
      $e,\ \boldsymbol{W}r(\xi) \leftarrow$ CalculateWeights(layer, r)
      $(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}),\ \boldsymbol{J}^T\boldsymbol{W}r(\xi) \leftarrow$ CalculateJacobian(layer, r, $\boldsymbol{W}r(\xi)$)

      $k \leftarrow 0$
      while true **do** ▷ Find best λ
        Solve $(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} + \lambda\text{diag}(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}))(\delta\xi) = \boldsymbol{J}^T\boldsymbol{W}r$
        $k \leftarrow k + 1$

        $\xi \leftarrow \delta\xi \circ \xi$ ▷ Update pose estimate
        $r \leftarrow$ CalculateResidual(layer)
        $e_{new},\ \boldsymbol{W}r(\xi) \leftarrow$ CalculateWeights(layer, r) ▷ $\boldsymbol{W}r(\xi)$ is unused

        if $e_{new} < e$ **then** ▷ Is estimate better?
          break
        else ▷ Try again with new value
          $\lambda \leftarrow v^k$ ▷ v predefined constant
        end if
      end while
    end for
  end for

  return ξ
end procedure

Kernel Name	Parallel Pattern	Calculation(s) performed
Calculate Residuals	Map	$r(\xi)$
Calculate Weight and Residuals	Map	$\boldsymbol{W}r(\xi)$
Calculate Jacobian	Reduce	$(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J})x = \boldsymbol{J}^T\boldsymbol{W}r(\xi)$
Solve	Ext. library	$(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} + \lambda\text{diag}(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}))(\delta\xi) = \boldsymbol{J}^T\boldsymbol{W}r$

Table 16.1: Kernels within the Tracking Building Block in LSD-SLAM

We tabulate the kernel timings in Table 16.2. The complete runtime figure captures the run time of the kernels and any extra logic included in the function, `trackFrame`.

	Non-vectorised		Vectorised	
Kernel	**Mean**	**Total**	**Mean**	**Total**
Calculate Residuals	0.0010432	27.147992	0.0009576	24.7622195
Calculate Weight and Residuals	0.0001945	10.551394	0.0000256	1.39351978
Calculate Jacobian	0.0002563	5.8821636	0.0001332	3.04969915
Solve	0.0000004	0.0163864	0.0000006	0.02550619
Total Run Time:	0.0161857	48.3569444	0.01136153	34.3763546

Table 16.2: The runtimes of kernel, within the Tracking Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward

16.1.4 Optimisations

The LSD-SLAM authors have provided a set set of optimised kernels, utilising data parallelism, for the 'Calculate Weight and Residuals' and 'Calculate Jacobian' kernels above.

The use of the map parallel patten, and also in this case for the reduce kernel enables easy porting to vector instructions. Here the authors have used SSE for x86 and NEON for ARM. Both of these instruction set extensions enable 128bit lanes (SSE enables wider lanes as well), which enables 4 single precision (32bit) floating point operations.

We can see in the timings in Table 16.2, which shows a large decrease in time per frame, of 40%!

Firstly, this is important for the front-end so that the master thread can utilise the poses, generated from the frames. But what is more interesting, is that the the enabling / disabling of the vectorisation barely makes a difference in the process every frame mode, this is because, as we will come to see, the processing time is now dominated by the depth map updating kernels. (A Kiviat plot comparing the two results is in the Appendix, Figure A.2.) This can be considered a problem with the process-every-frame mode. When we investigated the process-every-frame mode, we determined this was the best solution, as when the next frame is tracked, it has to tracked on the correct

key-frame, hence the need to wait for the depth-mapping thread to complete processing.

16.1.5 Further Commentary

Evidently here and throughout the implementation is that the system is somewhat self-regulating with respect to frame throughput. The `trackFrame` method is designed to track frames and notify the 'Depth Mapping' thread of all newly tracked frames, therefore the bulk of the work is performed by other threads. This enables the master thread to perform other tasks, possibly utilising the poses from the tracked frames. This is a real benifit of the parallel pipeline architecture.

16.2 Depth Mapping

The 'Depth Mapping' thread processes frames provided by the master thread. Frames are either used to update the key-frame's depth map (algorithm described in Section 7.3.4) or are promoted to key-frame status (algorithm described in Section 7.3.6). Moreover, the thread also handles the change of key-frame. The algorithmic outline of the implementation can be seen in Algorithm 6.

Algorithm 6 Mapping Iteration Thread

```
1: procedure MappingIterationThread()
2:     while keepRunning do
3:         DiscardOldFrames()
4:         frame ← GetLatestTrackedFrame()
5:         if tracking is good then
6:             if createNewKeyFrame then
7:                 changeKeyFrame(frame)
8:             else
9:                 updateKeyFrameDepthMap(key-frame, frame)
10:            end if
11:        else
12:            Perform re-localisation
13:        end if
14:        wait until new frame is tracked
15:    end while
16: end procedure
```

An interesting behaviour of this thread is how it handles 'old' frames (line 3, Algorithm 6). 'Old' frames are those which were tracked against a key-frame which has subsequently been inserted into the pose-graph, and therefore cannot be updated. These old frames are therefore ignored. This behaviour returns us to the probable purpose of LSD-SLAM, of controlling for example quadcopters. The auto-pilot software would always want a location update - provided by the tracking thread - but if this, the mapping

thread, is not able to process at the frame rate, it must gracefully handle this situation.

(When entering the 'Perform re-localisation' step the current LSD-SLAM implementation essentially halts operation, as it never recovers full SLAM operation. It does though keep accepting frames, but they are not used to update the key-frame or the pose graph.)

16.2.1 Core Methods

As can be seen in Figure 10.2, there are two possible directions a frame can take, based on its distance to the key-frame (discussed in Section 15.2.1). However, there are some shared kernels, which will be noted. For simplicity sake, we treat each path separately. Arguably there is a third, with re-localisation. However as the current implementation ceases to function, once the re-localiser is active, we don't consider this execution path.

16.2.2 Updating the Depth Map

This, the most frequently taken path (determined above in Section 15.2.1), updates the current key-frame's depth map by using epipolar geometry. There are three steps as shown in Algorithm 7.

The first step updates the depth map for the key-frame using the view provided from the new frame. The second and third methods clean-up the depth map, by removing erroneous depth measurements. They are a bi-product of using statistics for the semi-dense method.

Algorithm 7 Update key-frame's depth map

1: **procedure** UPDATEKEYFRAMEDEPTHMAP($key\text{-}frame, frame$)
2: OBSERVEDEPTH($key\text{-}frame, frame$)
3: FILLHOLESINMAP($key\text{-}frame$)
4: REGULARISEDEPTHMAP($key\text{-}frame$)
5: **end procedure**

The LSD-SLAM implementation makes use of parallel processing, namely (more) threads for these three methods. The implementation, blocks data by row, and assigns each thread a contiguous block. The kernels for this block are listed in Table 16.3, with timings listed in Table 16.4.

The kernel, 'Copy Depth Map to Frame', is an artefact from the implementation, we could not simply ignore its cost, therefore it is included. The first three kernels operate on a copy of the depth map from the current frame, this is to avoid data races. Hence, the kernel, 'Copy Depth Map to Frame', purpose is to copy back the resulting depth map from the first three kernels to the frame. Again there is a 'logic' overhead, per frame of 0.6 ms, in this case.

The Stereo Line Search function is mapped over all the pixels, which have not been flagged as 'bad', however it is a complex function. Its behaviour follows how we described in Section 7.3.4, when we discussed epipolar geometry. When LSD-SLAM is run within

Kernel	Parallel Pattern	Purpose
Stereo Line Search	Map*	Determines depth by mapping a function `doStereoLine` (which performs a search on the epiline, of a pixel, for an intensity) on all the pixels in a frame[3]
Fill Holes	Stencil	Fill in missing depth estimates, by interpolation from neighbours, where possible.
Regularise Depth Map	Stencil	Remove and reduce outliers from depth estimates.

Table 16.3: Kernels, for the purpose of depth estimation, within the Depth Mapping Building Block in LSD-SLAM

Kernel	Mean Time	Total Time
Stereo Line Search	0.00707	20.36965
Fill Holes	0.00330	9.4904
Regularise Depth Map	0.00466	13.402847
Copy Depth Map to Frame	0.00100	2.7115699
Complete Run Time:		**47.819**

Table 16.4: The runtimes of kernels, for the purpose of depth estimation, within the Depth Mapping Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward.

Intel VTune Amplifier[4] it identifies this function as one of the most time consuming functions in the program.

16.2.3 Changing the Key-Frame

The alternate path a tracked frame can take is to be promoted to key-frame status. Here the outgoing key-frame is finalised, and one of two paths are taken. Either the current frame takes depth information from the outgoing key-frame, or an existing key-frame is re-loaded, from the pose graph. The default behaviour is to always generate a new key-frame and never reuse existing key-frames.

We outline the structure of the implementation in Algorithm 8. Again we see this recurring pattern of `RegulariseDepthMap` and `FillHolesInMap`, (lines 3 and 4 of Algorithm 8). **SE**(3) to **Sim**(3) transformation is part performed by line 4, which enables the key-frame to be inserted into the pose graph, line 5.

The procedure `findCandidateFrame` is very similar to constraint searching, since the aim is to attempt to find the nearest frame - just like detecting loop-closures through

[4]Intel VTune Amplifier is a tool which enables performance tuning of software.

distance between frames.

On inspection of this algorithm, we were able to generate the kernels, outlined in Table 16.5.

Kernel	**Parallel Pattern**	**Purpose**
Propagate Depth Map	Map	Propagate a depth map from one key-frame to another using the relative pose between them.
Normalise Depth Map	Map and Reduce	Over all the pixels sum depth
Copy Depth Map	Map	Load depth map from existing key-frame.
Find Candidate Frame	Reduce	Find best key-frame to reuse. (The For loop from Algorithm 8)
Find Euclidean Overlaps	Search	Find nearby frames using Euclidean distance.
Overlap between frame	Map	Find approximate overlap between two frames

Table 16.5: Kernels, for the purpose of changing the key-frame, within the Depth Mapping Building Block in LSD-SLAM

To analyse the kernel timings, we split the processing into two parts: finalising the current key-frame (shown in Table 16.6), creating a new key-frame (shown in Table 16.7). Again, we have added an implementation specific kernel here, 'Copy Depth Map', which has the same functionality.

Kernel	**Mean Time**	**Total Time**
Fill Holes	0.0028	0.367
Regularise Depth Map	0.0043	0.563
Normalise Depth Map	0.0009	0.008

Table 16.6: The runtimes of kernels, for the purpose of finishing the current key-frame, within the Depth Mapping Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward.

Some of these times are not insignificant, however as we have determined in our discussion above (Section 15.2.1) that under default parameters, the key-frame path is infrequently taken, so the processing time does not on average effect the throughput significantly.

Algorithm 8 Algorithm outline for changing key-frames

```
1: procedure changeKeyFrame(frame)
2:     fillHolesInMap(key-frame)                          ▷ Finish Current key-frame
3:     regulariseDepthMap(key-frame)
4:     normaliseDepthMap(key-frame, frame)                ▷ Mean depth of 1
5:     addToPoseGraph(key-frame)

6:     bestFrame ← none
7:     if re-activate key-frame then          ▷ Search Pose graph for best new key-frame
8:         bestFrame ← findCandidateFrame(frame)
9:     end if

10:    if bestFrame = none then                           ▷ Create New Key-frame
11:        propagateDepthMap(key-frame, frame)  ▷ Propagate under projection
12:        regulariseDepthMap(frame)
13:        fillHolesInMap(frame)
14:        regulariseDepthMap(frame)
15:        bestFrame = frame
16:    else                                               ▷ Load existing key-frame
17:        extractDepthMap(bestFrame)
18:        regulariseDepthMap(bestFrame)
19:    end if
20:    current key-frame ← bestFrame
21: end procedure

22: procedure findCandidateFrame(frame)
23:    candidates ← findEuclideanOverlaps(frame)  ▷ Physically near-by frames
24:    candidates ← OpenFABMap(frame)             ▷ Similar frames by appearance

25:    bestFrame ← none
26:    bestScore ← inf

27:    for candidate-frame ← candidates do
28:        score ← overlapBetween(frame, candidateFrame)

29:        if score < bestScore then                      ▷ Smaller is better
30:            SE(3)Track(frame, candidate-frame)
31:            if candidateFrame.transformation < bestFrame.transformation then
32:                bestFrame ← candidateFrame
33:            end if
34:        end if
35:    end for

36:    return bestFrame
37: end procedure
```

Kernel	Mean Time	Total Time
Propagate Depth Map	0.00914	0.7865
Regularise Depth Map	0.00367	0.6832
Fill Holes	0.00371	0.3190
Copy Depth Map	0.00110	0.09469

Table 16.7: The runtimes of kernels, for the purpose of creating the key-frame, within the Depth Mapping Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward.

16.3 Constraint Search

The constraint search thread handles the first half of the pose graph optimisation, outlined in Section 7.3.6. The implementation constantly searches for constraints, either as key-frames are added to the pose graph or just picking randomly. The behaviour of this thread is outlined in Algorithm 9.

Algorithm 9 Constraint Search Thread

```
1: procedure ConstraintSearchThread()
2:     while keepRunning do
3:         if new key-frame then
4:             findAndAddConstraints(key − frame)
5:         else
6:             key-frame ← random key-frame from graph
7:             findAndAddConstraints(key-frame)
8:             sleep(500ms)                    ▷ Can be interrupted via signalling
9:         end if
10:    end while
11: end procedure
```

Constraint Search Function

The constraint search behaviour can be split into two halves:

1. Gathering candidate frames
2. Determining the best candidates, and therefore generating the constraints between the frames.

Firstly, to gather candidate frames, all key-frames in the pose graph are considered. Frames are chosen if they are suitably close or if they appear the same. Close frames are those which have a small translation between them. For appearance based comparison, this is passed to an external library, OpenFABMap. (This library is treated as a black

box). The candidates are split into two sets, close and far. Each set is sorted and only the best n overall are used in the next step.

Secondly, a 'reciprocal tracking check' is performed. This check makes sure the transformation between the two key-frames is similar i.e. from frame A to frame B and B to A. This is implemented as multiple **Sim**(3) tracking attempts, using the coarse to fine approach. By doing this one can avoid processing frames where they can be quickly rejected. This is performed so that incorrect constraints are not added to the graph - or at least very unlikely to be. Reciprocal tracking has to use **Sim**(3) tracking since the key-frames from the graph include scale. The outline for the algorithm is shown in Algorithm 10.

Algorithm 10 Find constraint method in constraint search thread

```
1: procedure findAndAddConstraints(frame)
2:     candidates ← findEuclideanOverlaps(frame)   ▷ Physically near-by frames
3:     candidates ← OpenFABMap(frame)              ▷ Similar frames by appearance

4:     Filtering and sorting logic for candidates

5:     for candidate - frame ← candidates do
6:         if reciprocalTrack(frame, candidate) then
7:             Add constraint
8:         end if
9:     end for
10: end procedure
```

The **Sim**(3) tracking between key-frames utilises much of the same code structure as **SE**(3), the only significant difference being the change in Jacobians. Therefore, there is one kernel we can extract from the constraint finding algorithm outlined in Algorithm 10, and four from **Sim**(3) tracking. This is shown in Table 16.8. We present the timings in Table 16.9.

Kernel	Parallel Pattern	Purpose
Find Euclidean Overlaps	Search	Find the nearby frames.
Calculate Residuals	Map	$r(\xi)$
Calculate Weights as Residuals	Map	$\boldsymbol{W}r(\xi)$
Calculate Jacobian	Reduce	$(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J})x = \boldsymbol{J}^T\boldsymbol{W}r(\xi)$
Solve	Ext. library	$(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J} + \lambda\text{diag}(\boldsymbol{J}^T\boldsymbol{W}\boldsymbol{J}))(\delta\xi) = \boldsymbol{J}^T\boldsymbol{W}r$

Table 16.8: Kernels within the Constraint Search Building Block in LSD-SLAM

Kernel	Mean Time	Total Time
Find Euclidean Overlaps	0.00459	1.165
Filter and Sorting	0.00278	0.691
Calculate Residuals	0.00014	13.47
Calculate Weights as Residuals	0.00003	1.371
Calculate Jacobian	0.00010	2.234

Table 16.9: The runtimes of kernels within the Constraint Search Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward.

16.4 Optimisation

The final stage is optimisation, whose purpose is to improve the quality and correctness of the pose graph of key frames, by finding the "best parameters which explain the set of measurements" [71]. The majority of the computation is performed by the g2o library, which was designed for solving this exact problem. We treat g2o as a black box, but in Algorithm 11 we show how it is integrated.

Algorithm 11 Optimisation Thread

```
1:  procedure OptimisationThread()
2:      while keepRunning do
3:          wait until new new constraint
4:          if doFinalOptimisation then                    ▷ No frames remaining
5:              optimisationIteration(50, 0.01)
6:              return
7:          end if
8:          until fails do optimisationIteration(5, 0.02)  ▷ Run until no changes
9:      end while
10: end procedure

11: procedure optimisationIteration(itsPerTry, minChange)
12:     Run g2o optimisation                               ▷ Trying itsPerTry times
13:     for all frames ∈ FrameGraph do                     ▷ Save optimisation calculations
14:         update pose
15:     end for
16: end procedure
```

Since it is a black box, we do not extract any kernels, however, we have recorded the time spend processing, shown in Table 16.10.

The dataset we have been operating on for this timing analysis, TUM RGB-D fr2/xyz, only has one loop closure, so we are really not testing the full capabilities of g2o here.

Timing Block	Mean Time	Total Time
g2o call	0.0136	3.395
Update graph after optimisation	0.0001	0.035

Table 16.10: The runtimes of the timing blocks within the Optimisation Building Block in LSD-SLAM. Using default parameters under TUM RGB-D fr2/desk, on Seyward.

16.5 Summary

In this Chapter decomposed LSD-SLAM into kernels, which has enabled an understanding of the inner workings. Furthermore, we have also isolated the core kernels, such as 'Stereo Line Update', which is where the current bottleneck lies.

We have seen the effect of optimising the **SE**(3) tracking code, but this highlighted an issue with the process-every-frame mode, since this approximately 50% improvement was not being realised.

By extracting the kernels we have laid the groundwork for future investigations where they can be extracted, or swapped with other algorithms or implementations.

Chapter 17

Design Space Exploration of LSD-SLAM

Using our understanding of how LSD-SLAM operates, at the building block and kernel level, we now turn our attention to specifically exploring the parameter design space. This will enable us to see what trade-off's are possible with respect to the three SLAMBench metrics.

17.1 Methodology

We compose the design space of the parameters mention in Table 10.4. This is clearly a large space, and therefore cannot be exhaustive searched, within a reasonable time frame. Moreover, searching for optimal parameters, with in SLAMBench, is not necessarily advantageous as they will suit one particular dataset and possibly not generalise for all datasets. We therefore select a select a few potentially interesting regions to explore, commenting on the effects and trade-off's made.

17.2 Sparsity Control Parameters

Since, in this report we are focusing on semi-dense verses dense techniques, it is worthwhile investigating how altering sparsity related parameters affects the three metrics.

We investigate the how LSD-SLAM selects pixels in order to calculate depth (the 'Minimum Gradient Threshold' Section 17.2.1), along with when it decides to promote frames to key-frames (Section 17.2.2).

17.2.1 Minimum Gradient Threshold

The minimum gradient cut-off, (accessible via the `--minusegrad` command-line option), sets a threshold which filters out those intensity gradients which are too shallow, thereby using a smaller set of the available pixels. This takes place before a pixel is used to update a depth measurement. (See Section 7.2.1).

Gradients in the frame can be caused by a primarily two factors: contrast between objects and noise. For example, consider a computer monitor on a desk and the scene behind (Figure 17.1), this will have around the monitor edge a change in colour, between the monitor bezel and the background. When converted to monochrome, this will be realised as an intensity difference, therefore a gradient can be calculated. This could be a very steep gradient, however there will be a range of gradients throughout the scene.

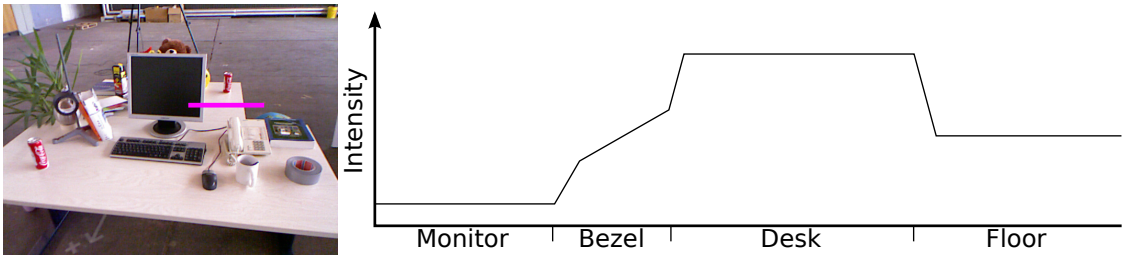


Figure 17.1: Diagram visualising an intensity gradient. The graph shows an approximate rendering of the intensity gradient along the pink line in the image. (A frame from TUM RGB-D fr2/desk)

To demonstrate, and understand what effect limiting the minimum usable gradient, we evaluated at every integer value within the suggested[1] minimum gradient threshold range, $[1, 50]$. Our results are plotted in Figures 17.2, and 17.3.

We can gather two important observations:

1. Difference between real-world and synthetic datasets due to lighting differences.
2. A trade-off between FPS and ATE.

The difference between the (synthetic) ICL-NUIM Living Room and the (natural) TUM RGB-D datasets is very clear. There is far less variation in the intensity gradients in the ICL-NUIM Living Room frames. This could be caused by the minimal use of textures in the environment, but equally the simple lighting. The natural dataset frames are from a very 'busy' scene, so there are lots of usable intensity gradients. An argument could be made that the ICL-NUIM Living collection should not be used due to its synthetic nature however it does provide exact ground-truth measurement and avoids some of the issues arising with using natural datasets (e.g. the asynchronous data-capture.)

Moreover, notice in Figure 17.3 that the ATE reduces as the minimum gradient threshold is increased. This is somewhat unexpected. However, I suspect that this is caused by improving the signal to noise ratio, by filtering out low quality gradients. But it still eventually fails to track.

The more interesting observation is trade-off between FPS and ATE. Firstly, recall, from Figure 15.7, that the most common path is through depth estimation and key-frame update methods. Secondly, consider the behaviour of the general depth-estimation kernel

[1]Based on the values at `https://github.com/tum-vision/lsd_slam/blob/master/lsd_slam_core/cfg/LSDParams.cfg`. We have removed this file our version of LSD-SLAM as it is ROS.

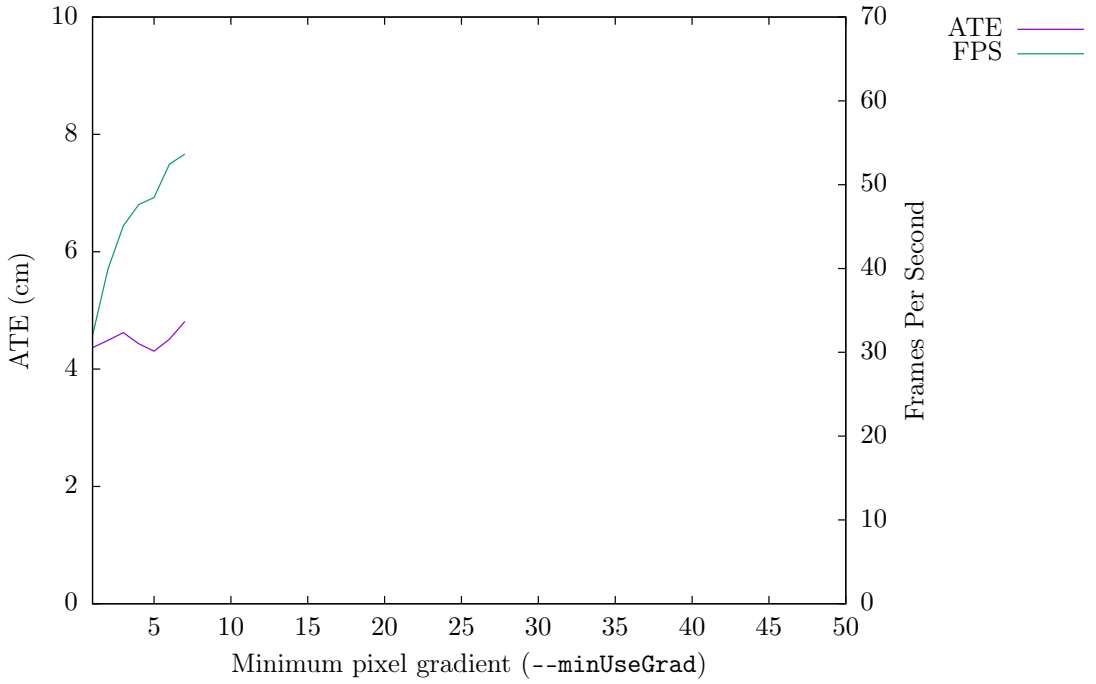


Figure 17.2: ATE and FPS using LSD-SLAM running on ICL-NUIM Living Room trajectory 2 dataset changing minimum gradient threshold. (Run on Seyward with default parameter, except the minimum gradient threshold)

(shown in Algorithm 12). It is clear by increasing the minimum bound on the threshold, will reduce the number of `PerformComplexOperation`'s will be called. Therefore, the FPS will sky-rocket as the common path time decreases, when there are fewer pixels to compare.

Algorithm 12 General Depth Estimation Kernel

```
1: for h in height do
2:     for w in width do
3:         if GradientAt(h, w) < Minimum Usable Gradient then
4:             Invalidate Pixel
5:             continue
6:         end if

7:         PerformComplexOperationAt(h, w)

8:     end for
9: end for
```

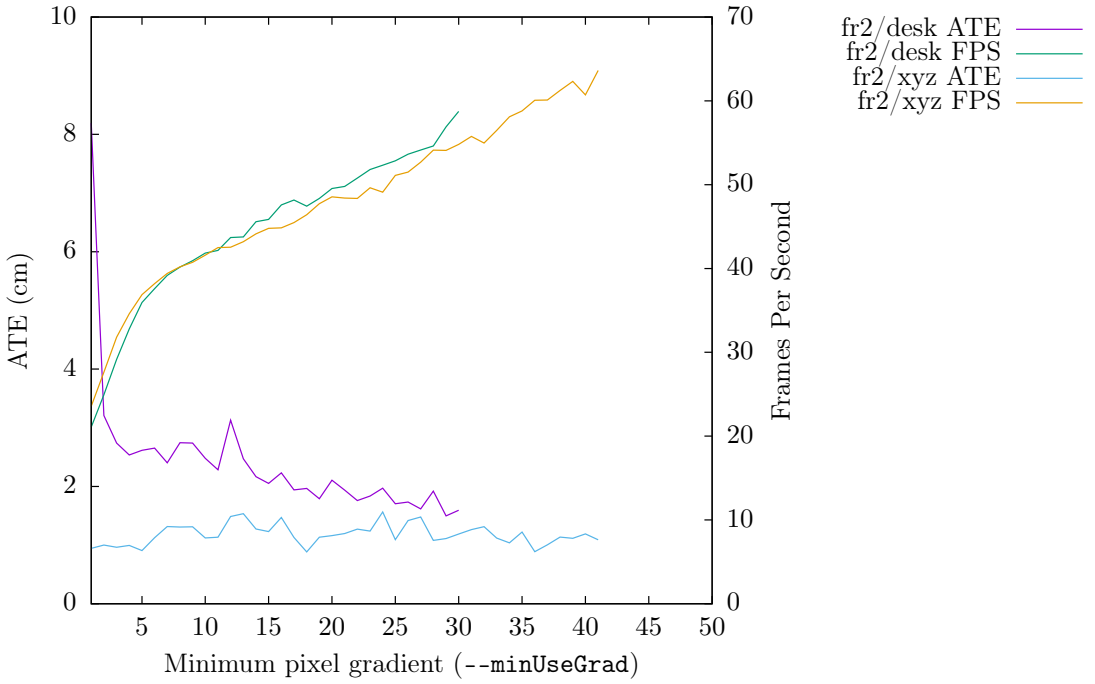


Figure 17.3: ATE and FPS using LSD-SLAM running on TUM RGB-D fr2/xyz and fr2/desk datasets changing minimum gradient threshold. (Run on Seyward with default parameter, except the minimum gradient threshold)

17.2.2 Frame Promotion

In our second of the design space explorations on sparsity, we investigate the parameters for controlling when key-frames are picked.

Frames are selected to be promoted based on two distance parameters. The first takes the euclidean distance between the two frames (using just the translation) and also the number of shared points between them.

We have explored the full design space of these two parameters, the bounds, $(1, 20]$, of which are suggested in a configuration file.[2] We have run this exploration on the TUM RGB-D fr2/xyz, TUM RGB-D fr2/desk and ICL-NUIM Living Room trajectory 2 datasets. We show the results of TUM RGB-D fr2/desk here, in Figures 17.4 17.5 17.6, as this enables the most interesting commentary. The other results are in the Appendix, Section A.3.1.

As we can see, where LSD-SLAM can maintain tracking for most points within the design space, and there is minimal variation in the ATE and FPS (Figures 17.4 17.5 respectively). The edges around the tracking failures, show 'interesting' results, in that the the algorithm can perform very well (e.g. at point (5,1)) and poorly (e.g. (1,5)).

[2]Viewable here: `https://github.com/tum-vision/lsd_slam/blob/master/lsd_slam_core/cfg/LSDParams.cfg`

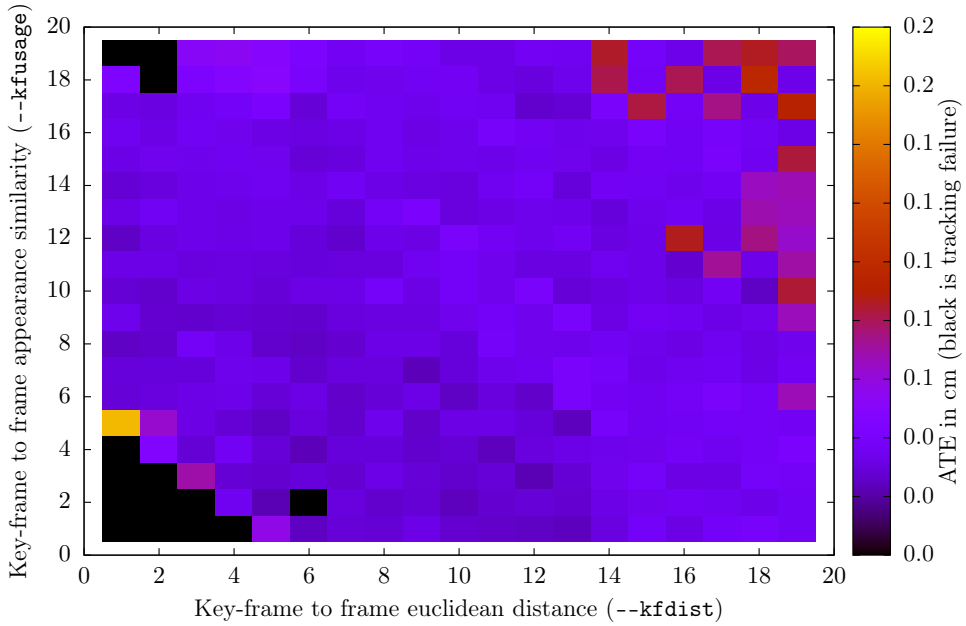


Figure 17.4: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/desk on Seyward with otherwise default parameters. This colours represent ATE (cm).

Effect on Pose Graph

One aspect we have not discussed in detail is the final iterations of the constraint finder and optimiser. This takes place after all the frames have been accepted. Its purpose is to find the remaining constraints (loop closures) and improve the pose estimates of the key-frames. We can see that varying the key-frame thresholds greatly effects the finalisation time. We show this in Figure 17.6 where LSD-SLAM was run with TUM RGB-D fr2/desk. (Again, see Appendix, Section A.3.1 for results with some other datasets.)

In order to understand this behaviour, we highlight a few key points in the design space, noting the number of constraints and key-frames at each point. This is shown in Figure 17.7.

We take each of the four highlighted points in turn.

Top Left (3,19)
We can see that 7% of frames are being promoted to key-frame status, and there are a large number of constraints. By doing this, LSD-SLAM is forcing g2o to provide accurate pose estimates. This is possibly 'cheating' as this has now become an off-line problem, solved once, and not solved continuously.

Top Right (19,19) / Bottom Right (19,1)
Even though, at the top right, this is the extreme for both parameters, there are fewer constraints than the top left corner, but many more key-frames (10% frames promoted).

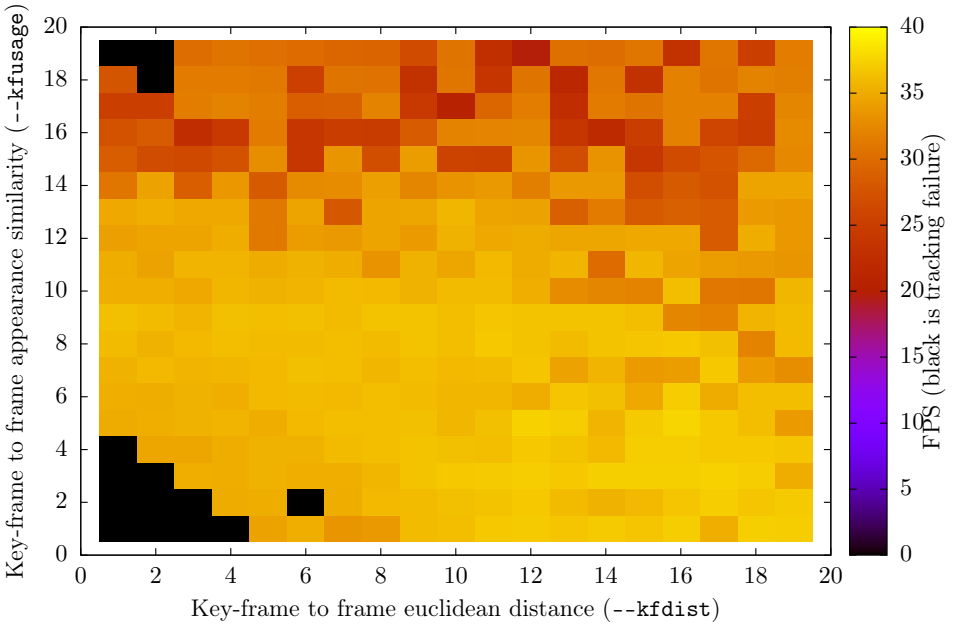


Figure 17.5: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/desk on Seyward with otherwise default parameters. This colours represent FPS.

The reason for this is that the euclidean distance parameter is also used to limit the search radius for determining which frames to consider for constraint finding. The bottom right corner is consistent with this finding.

Default (5,5)
This seems to be a happy medium between delaying pose optimisation, and keeping it in near real-time.

ATE Without Finalisation

We have seen that for certain regions of the design space, the finalisation time is large. For the highlighted points above we have graphed (shown in Figure 17.8) the trajectory without this finalisation step. We can see that the calculated trajectory is a long way off for the cases where there are large number of key-frames. This shows that the loop-closure detection and optimisation step is required, but also that it has a great effect.

Furthermore, we can see that there is a clear trade-off, by delaying the pose optimisation till last, in that the real-time pose estimates will be a long way off.

Conclusion

We have explored the entire design space for the two primary variables in selecting a key-frame. We have seen how they affect, primarily, the finalisation time. Also very

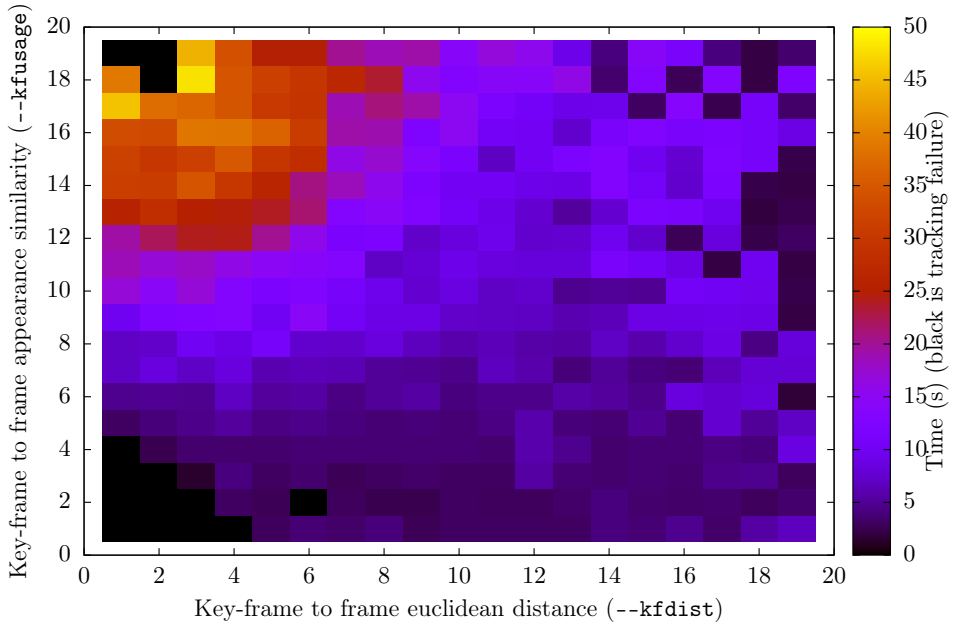


Figure 17.6: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/desk on Seyward with otherwise default parameters. This colours represent finalisation time.

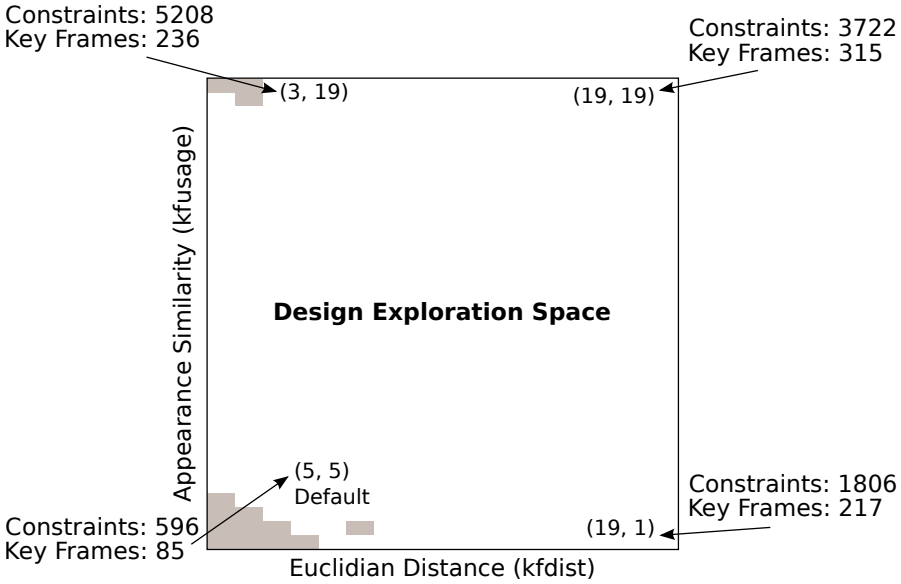


Figure 17.7: The design space for key-frame selection (`kfusage`, `kfdist`), highlighting some key points, noting the total number of constraints and key-frames. The results are taken from LSD-SLAM running on Seyward using the TUM RGB-D fr2/desk dataset.

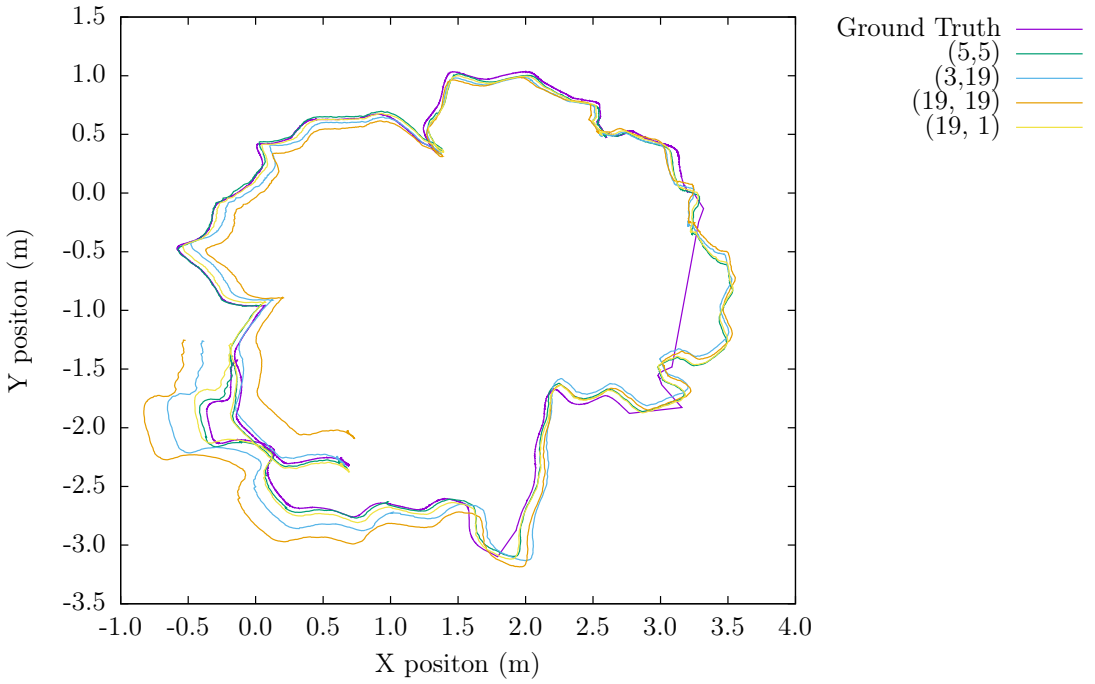


Figure 17.8: Comparing the calculated trajectories of the highlighted points of the design space in Figure 17.7. These trajectories have not had final pose optimisation applied. (LSD-SLAM running on Seyward using the TUM RGB-D fr2/desk dataset.)

low values causes LSD-SLAM to loose tracking, therefore higher values are better. The default seems to be suitable.

17.3 Hardware Parameters

We have already seen a little of the effect of changing the hardware platform, but so far we have not investigated this, nor altered the settings of the platforms. We now investigate a few more trade-offs which can be had, with regards to the hardware platform. We investigate the following:

1. CPU frequency on Seyward
2. Availability of cores on the ODROID

We take each of these in turn.

17.3.1 CPU Frequency on 'Seyward'

The results up to now on Seyward have relied on the CPU to govern it own frequency, and scale it appropriately. We now perform a set of experiments to investigate the effect

of fixing the CPU frequency. As LSD-SLAM is deterministic, it is unnecessary to plot the ATE, but clearly the CPU frequency will affect the FPS and energy used per frame. Our results are shown in Figures 17.9, 17.10, 17.11.

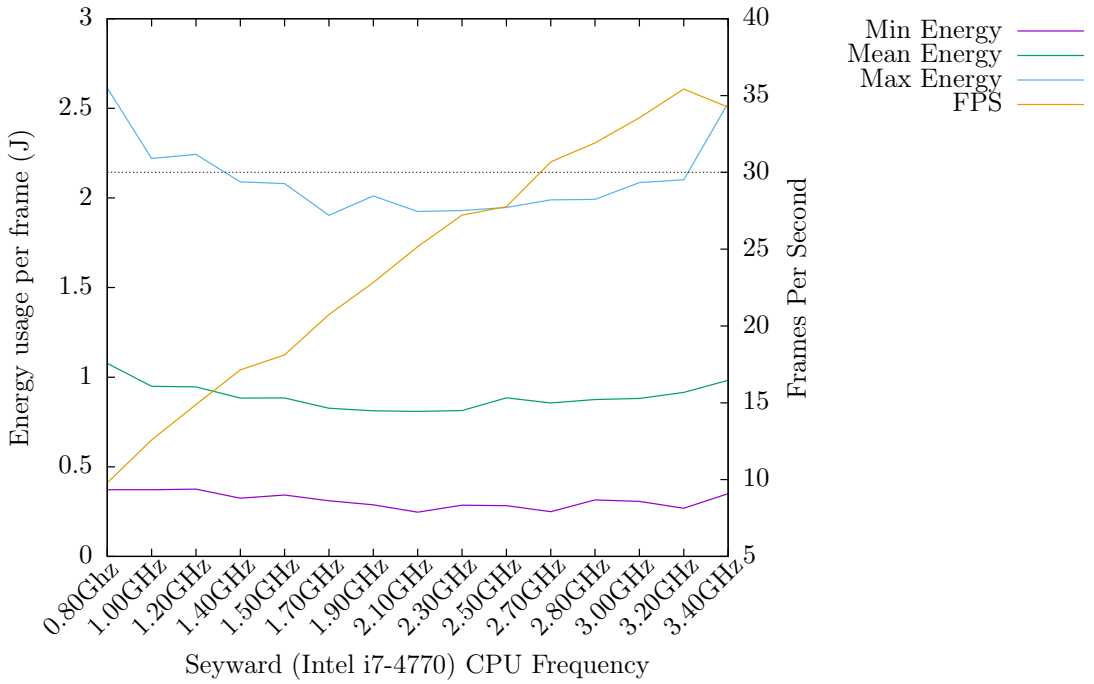


Figure 17.9: FPS and energy per frame of LSD-SLAM whilst changing CPU frequency, on Seyward, using TUM RGB-D fr2/xyz.

These results again highlight LSD-SLAM's dependency on the input characteristics. The minimum threshold, we defined earlier for FPS, at 30 FPS is reached sooner i.e. at a lower frequency than both the TUM RGB-D fr2/xyz and fr2/desk datasets. This is because there are fewer points with a suitable intensity gradient in the synthetic ICL-NUIM Living Room Traj. 2 dataset compared with the other 'real world' datasets.

17.3.2 Availability of the Processing Cores on the ODROID

When we introduced the hardware platforms, we noted that the ODROID featured a 'big.LITTLE' architecture. This means there are two distinct processor groups, in this case four ARM Cortex-A7 cores and four ARM Cortex A15 cores. The Kernel, together with the hardware manages which processes are executed on which of the cores.

We have already noted that moving from the Seyward to the ODROID, there was a huge trade-off of FPS for substantial reduction in energy.

The optimisations found in the x86 version of LSD-SLAM can be found in the ARM version, which utilises NEON SIMD instructions instead of SSE for x86.

In Figures 17.12, 17.13, we graph the effect of changing the processor availability, on

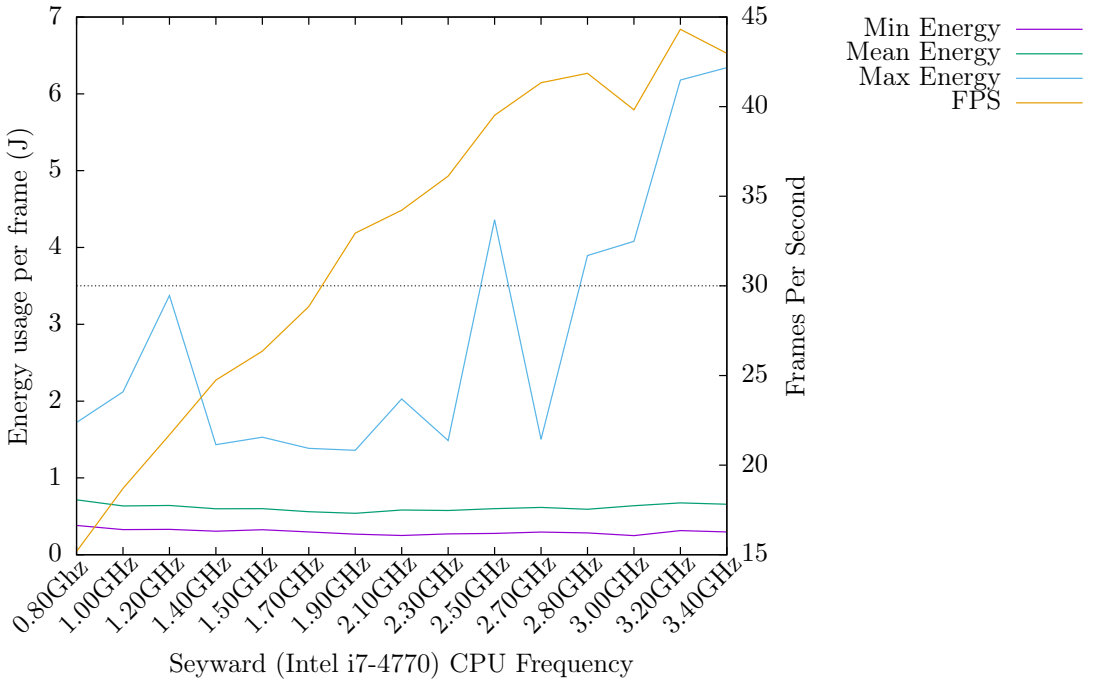


Figure 17.10: FPS and energy per frame of LSD-SLAM whilst changing CPU frequency, on Seyward, using ICL-NUIM Living Room Traj. 2.

the three metrics, for both LSD-SLAM and KFusion. (There are more plots in Appendix A.3.2, with very similar results.)

We can see both algorithms trade off a reduction in the energy required pre frame for FPS. However, LSD-SLAM is 'better'. Its energy per frame, with both the Cortex-A15 and Cortex-A7 or just the Cortex-A15, is significantly less, approximately 8× than KFusion. Moreover, it is achieving 4 FPS, which is still not anywhere near our threshold of 30 FPS for stating real-time behaviour.

17.3.3 Achieving Real-time Performance on Embedded Devices

We have seen that in its default state, LSD-SLAM on the ODROID cannot operate in real-time. It is worth noting that to obtain real-time tracking, there are quite a few changes to LSD-SLAM required. Schöps *et al* achieved real-time performance on an embedded device by essentially only performing tracking, with that at a greatly reduced frame resolution to 160×120. They also do not perform depth-mapping, loop-closure detection or pose optimisation, in their real-time augmented reality mode [14].

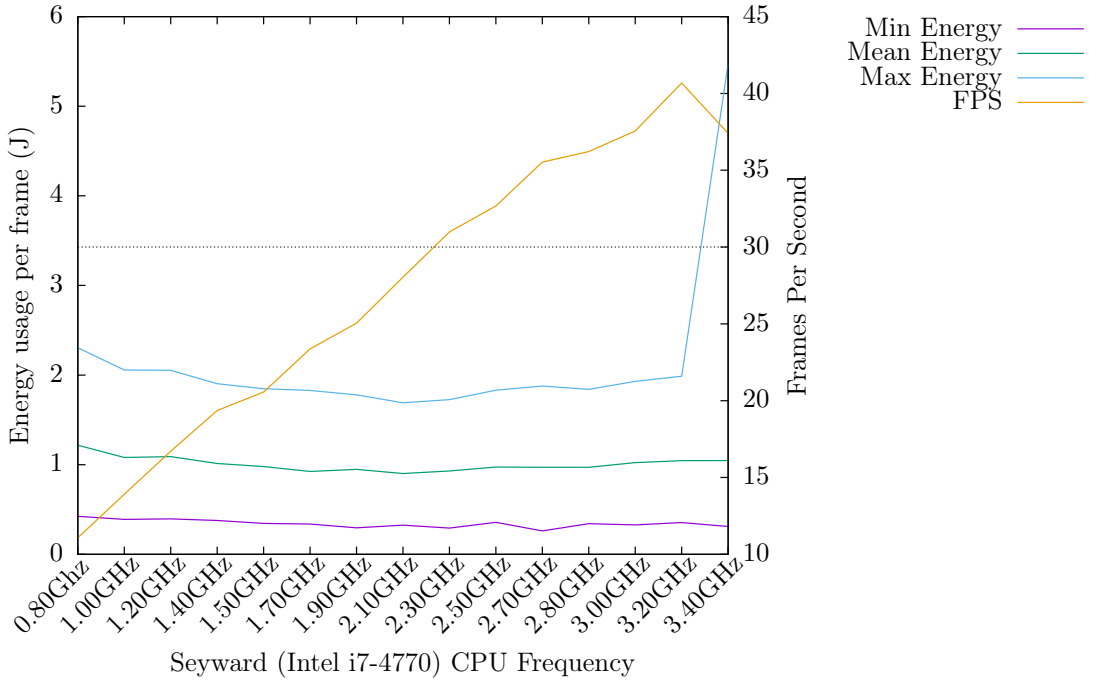


Figure 17.11: FPS and energy per frame of LSD-SLAM whilst changing CPU frequency, on Seyward, using TUM RGB-D fr2/xyz.

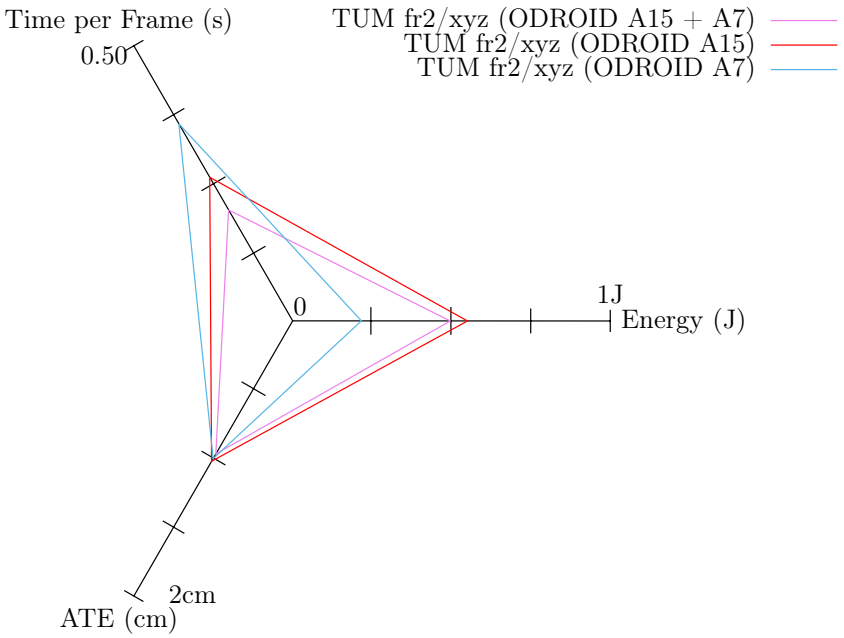


Figure 17.12: LSD-SLAM running on the ODROID, using default parameters on the TUM RGB-D fr2/xyz dataset. We have varied the availability of processing cores within the big.LITTLE architecture.

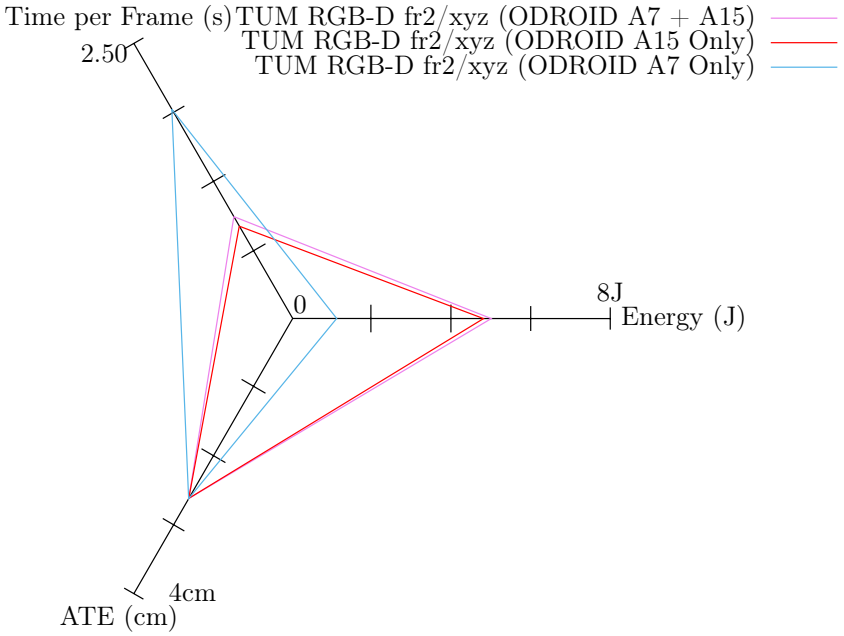


Figure 17.13: KFusion running on the ODROID, using default parameters on the TUM RGB-D fr2/xyz dataset. We have varied the availability of processing cores within the big.LITTLE architecture.

Chapter 18

LSD-SLAM in the Wild

So far in this investigation into 'Semi-Dense and Dense 3D Scene Understanding in Embedded Multicore Processors' we have investigated within the confines of deterministic behaviour, however this does not allow use to appreciate the full utility of a parallel pipeline. We will disable deterministic behaviour, but we will still be operating within the SLAMBench framework, so we can still gather and analyse the three metrics: ATE, FPS and Energy.

When we integrated LSD-SLAM into SLAMBench we 'tied' some of the threads together, in order to have deterministic behaviour, however this is not how LSD-SLAM was intended to be used. It was designed to, and will have throughout this section, four fully asynchronous threads - the building blocks - shown in Figure 10.2.

18.1 Required Changes to Methodology

In order to investigate how LSD-SLAM can operate in the real world, which, as we have mentioned, we have to leave determinism behind, this means we also have to disable the process-every-frame mode.

18.1.1 Input Frame Selection

Under this non process-every-frame mode, we need to simulate a real input, where frames are provided at a fixed rate.

The most straightforward way to implement this is to operate in pull model, rather than a push model. That is, the master thread requests frames when it is ready. The back-end frame provider will calculate the correct frame to provide given the change in time, since the last request. Therefore frames could be skipped, if the tracking / master thread cannot track above the input frame rate.

18.1.2 Trajectory Reconstruction

The skipping of frames lead to a problem: the algorithm has no knowledge about the un-tracked frames and therefore no pose has been determined for them. This causes

issues when calculating the trajectory error.

We have assessed two techniques to solve this problem. They both record the belief the algorithm has at all tracked frame, but they vary in their technique for comparison. The techniques were:

1. Only compare the poses of tracked frames.

2. Any untracked frame assumes the pose of the last tracked frame[1]

The first method will possibly lead to incorrect results. Consider a case when only one frame is accepted and tracked perfectly. The results will suggest a perfect SLAM algorithm - which is currently not obtainable. Moreover, it could be possible for some algorithm and dataset combination in which the algorithm performs very well when skipping some frames.

The second method, possibly better represents the lack of knowledge. The dataset trajectory has moved on but the algorithm is still believing it has not as it has not caught up with the latest frames - hence the ATE should encode this failed belief. This is the method we use (and this is what KFusion, in SLAMBench uses, when it cannot track a particular frame).

18.2 Frame Rate

The primary parameter, now is the input frame-rate. We mentioned in Section 13.2.1, the 'Simplification Assumptions', that we target a frame-rate of 30 FPS, as this is the upper bound on the output frame rate of the Kinect Camera. Therefore, to understand the effect of this, we vary the input frame-rate to LSD-SLAM, in unit steps in the range $[1, 30]$. The results are presented in Figures 18.1, 18.2.

In both Figures (18.1, 18.2), we can see the Seyward can process consistently at 30 FPS, this is expected as in the process-every-frame mode it was handling greater than this rate.

However, the ODROID shows its ability to process at a higher throughput, albeit with a significant trade-off in the ATE. Though it still does not reach the desired 30 FPS. Moreover, with this non-deterministic mode, tracking can fail, due to some adverse processing order. LSD-SLAM on the ORDROID failed at a range of different frame rates above 18 FPS, which we consider as unreliable, therefore we stopped plotting the results.

18.3 Conclusion

We can see from this that the parallel architecture enables processing of different input frame-rates without any configuration changes.

The primary benefit is being able to track frames quickly, with the (depth) map update taking place asynchronously.

[1]The first frame is always tracked so the first, and subsequent poses will be the scene coordinate frame and not some invalid location.

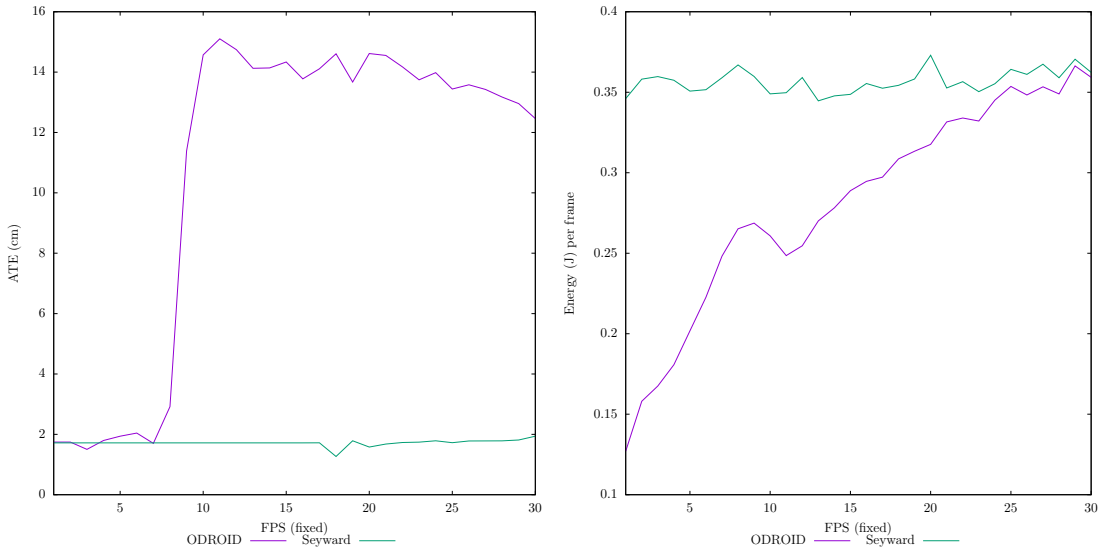


Figure 18.1: Varying the input frame-rate in LSD-SLAM under the TUM RGB-D fr2/xyz dataset. (Not using process-every-frame mode.) Shows ATE and energy under Seyward and ODROID.

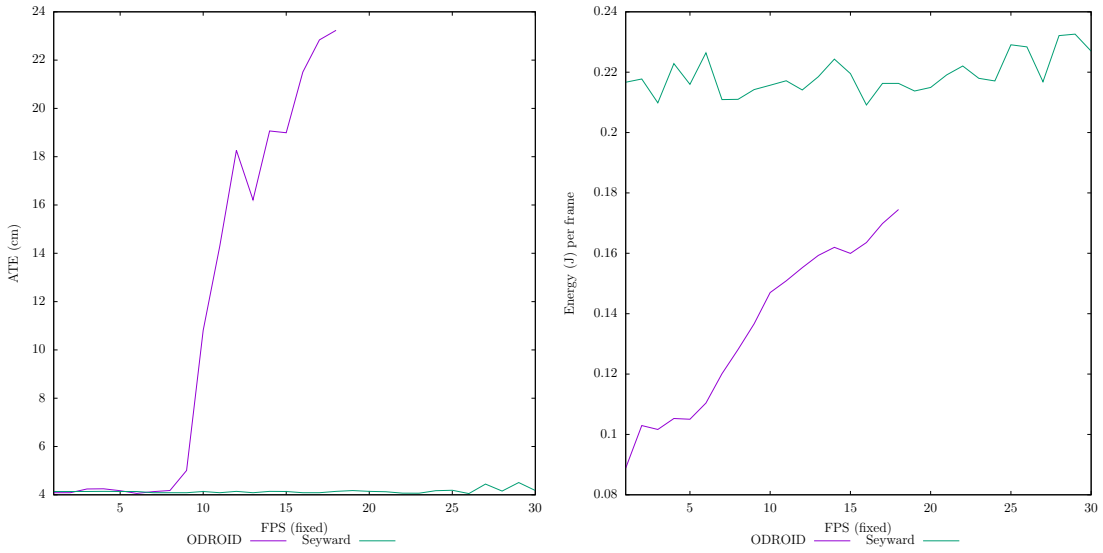


Figure 18.2: Varying the input frame-rate in LSD-SLAM under the ICL-NUIM Living Room trajectory 2 dataset. (Not using process-every-frame mode.) Shows ATE and energy under Seyward and ODROID.

Also LSD-SLAM can still maintain tracking, though at a cost of a larger ATE. We have seen a slight increase in the FPS handling rate on the ODROID, but could be more if we increased the limit on a suitable ATE.

Chapter 19

Comparison Evaluation and Summary of Results

To conclude our investigation comparing a dense and sequential SLAM algorithm (KinectFusion) with a semi-dense and parallel algorithm (LSD-SLAM), we highlight some of the key observations we have made.

19.1 Methodology Evaluation

We begin by evaluating our methodology, which will put into context our observations.

By using the SLAMBench framework we have been able to perform analysis across a variety of devices, and datasets. This was primarily made possible by the deterministic process-every-frame mode enforced by SLAMBench. However, throughout our investigation we noted a few issues with our extension to the SLAMBench methodology, for the purposes of comparing these two algorithms. We now highlight these concerns.

Exhaustive Design Space Exploration

We emphasised the need to perform an exhaustive design space exploration to validate all results for comparison. We only explored a small subset of the design space, which may not include any areas of optimal performance (in any/all of the three metrics).

Choice of Algorithms

The choice of algorithms, may in hindsight, might not have led to the fairest of comparisons. Primarily, because KinectFusion was designed with the idea of dense scene reconstruction, where as LSD-SLAM was designed to primarily preform (visual) odometry. However, the comparison is still interesting, even if at minimum it highlights the need for a wide variety of datasets, including non-synthetic ones.

We partially attend to this concern when we look at the future work, Section 20.2.

Defining Failure

I might have been too harsh in stating that a single untracked frame constitutes failure. Although, the LSD-SLAM algorithm, in its current state, gives up, we could have ignored the failure and continued accepting frames. This may have improved the 'interesting results' in Figures such as 17.2 and 17.3, where there is sharp failure rather than a worsening of the ATE, before failure.

Level of Optimisation

We have also noted how the comparison between different algorithmic implementations is not necessarily fair, given a different level of optimisation. To rectify this would require a large software engineering effort to when integrating SLAM algorithm implementation so all algorithms will have a similar level of optimisation, and hardware usage (e.g. both algorithms utilise a GPGPU.)

19.2 Sparsity

We now summarise our results with respect to the sparsity:

1. We have seen how dense methods are predicable with respect to the amount of work they perform. However, in KinectFusion's case this is large due to the mapping structure - the TSDF volume. In LSD-SLAM's case this was variable, and depended on the number of suitable pixels it could use (those where there is a suitable intensity gradient).

2. Furthermore, we have highlighted how it is consistent with the ATE's as a percentage of the room size. We have shown how it is robust across a variety of datasets, but it can perform better in certain scenarios. As a limit of KinectFusion, but not necessarily of dense algorithms in general is that it cannot cope at arbitrary scale, due to the requirement of knowing the scene size (for the TSDF volume initialisation) before commencing.

3. One the other hand, sparser methods, like those found in LSD-SLAM we have shown that they can out-perform dense methods in all the metrics we have investigated, but can also completely fail to track (in the case of the ICL-NUIM Office Scene.)

4. Furthermore, we showed how although on average the ATE is better, both algorithms can suffer when they are presented with datasets which they are less suited to. We looked specifically at the spread of tracking errors, which can vary wildly.

5. We delved deep into the aspects causing tracking failure, especially with the ICL-NUIM datasets. The cause was how it selects pixels to calculate depth estimates. This makes LSD-SLAM, and in general any SLAM algorithm utilising intensity gradients, very susceptible to lighting conditions and the textures within the scene.

We performed a design space exploration around the minimum gradient threshold, which highlighted a trade-off between ATE and FPS.

19.3 Parallel Architecture

As a second thread of investigation, we analysed the parallel architecture of the two algorithms at the pipeline level and lower down at the kernel level, though building blocks and parallel pattens, respectively. Below, we summarise our results with respect to the parallel behaviour:

1. The parallel pipeline of LSD-SLAM has brought to our attention the utility of a asynchronous parallel architecture. We saw work can be 'hidden' by off-loading it so that tracking can take place at speeds above frame rate. Parallel techniques have good applications for the real-world, as we mentioned previously about the application of LSD-SLAM for quadcopters.

2. Moreover, an *asynchronous* pipeline, enables it to be 'self regulating,' whereby if the depth-mapping and other components cannot keep up with the tracked frames, they can ignore the tracked frames, and not block the tracking. This result was particularly prevalent when operating in the 'real world', i.e. with a fixed input frame rate.

3. From the sequential pipeline of KinectFusion, we have seen that along with the dense SLAM method it provides a predictable performance, but as was indirectly shown in the SLAMBench paper, parallel implementations of the individual building blocks is required for any reasonable performance levels.

19.4 Similarities

We noted how both of these algorithms are similar in that for tracking they assume the map is perfect, then to update the map they assuming the tracking was perfect. Moreover, in both cases the map updating is one of the most expensive parts of the algorithms (this is especially true with KinectFusions, 'Integration' kernel.).

Moreover, both implementations cannot perform real-time on an embedded platform, using the CPU only. However, LSD-SLAM can track a slightly higher frame rate.

Neither of these state-of-the-art algorithms are perfect, and there is still research required to solve the SLAM problem!

Part V

Conclusion

Chapter 20

Conclusion

In this final chapter, we bring to a close our investigation into 'Semi-Dense and Dense 3D Scene Understanding in Embedded Multicore Processors'. We firstly cover our main achievements and close with a suggested path for future work.

20.1 Summary of Achievements

We began this project, with the aim of investigating the differences between semi-dense and dense methods, as well as parallel and sequential methods in the context of SLAM.

To this end, we focused on the first half of this report, integrating a semi-dense and parallel algorithm, LSD-SLAM, into SLAMBench to complement the existing, dense and sequential, algorithm KFusion.

1. We integrated LSD-SLAM into SLAMBench, taking into consideration the SLAMBench requirements, which meant we had to investigate the legal issues as well as provide a deterministic process-every-frame mode.

2. Further to this we also extended SLAMBench's dataset support to the TUM RGB-D dataset collection, including providing the tools to automate the three metrics evaluation under this dataset.

By using this integration, we were able to compare and contrast the two algorithms, both in terms of their sparse and parallel nature.

1. We investigated both algorithms at three levels: the algorithm as a single unit, the building blocks and finally the kernels, which provided us with insight into the predictability of a dense and sequential algorithm - KFusion. But, we also showed how LSD-SLAM, in some circumstances (particularly in some TUM RGB-D datasets) can out perform KFusion, in all three metrics (ATE, Energy and FPS).

2. One of our most interesting results, was the dependence on the dataset. This was highlighted through the histogram graphs showing the spread of results. This

lead us two two conclusions. Firstly, that a wide range of datasets are needed when evaluating SLAM algorithms. But, secondly, it enabled us to see particular differences in the algorithm operations. For example we highlighted the quantity of work for LSD-SLAM is based directly on the textures in lighting within the scene. This is reflected in primarily in the FPS and energy metrics, but also in the ATE.

3. Though this work did highlight the need to perform an exhaustive design space exploration in order to verify our results.

4. Furthermore, we performed some design space exploration to further investigate trade-offs between the metrics, particularly focusing on the sparsity parameters. We showed how a trade-off can be made between the ATE and FPS by altering the minimum gradient threshold. Moreover, we showed that by taking more key-frames, a trade-off can be made between FPS and the final pose graph optimisation step.

5. Penultimately, we investigated some hardware parameters. We compared the performance between a desktop processor (x86) and an embedded platform, the ODROID (ARM Cortex-A7 and Cortex-A15). This highlighted that there is a large reduction in the energy used per frame, and also the algorithms are both a *long* way from real-time performance on embedded devices.

6. We concluded this work to see how LSD-SLAM can perform without being confined to a process-every-frame mode, but rather with a fixed input frame rate. This which highlighted the utility of a parallel, asynchronous pipeline, particularly in the trade-off between ATE and FPS.

I believe we have been successful in investigating 'Semi-Dense and Dense 3D Scene Understanding in Embedded Multicore Processors,' by providing analysis and insight of two differing SLAM techniques, but also extending the SLAMBench framework.

20.2 Future Work

There are a variety of directions which can be taken, continuing from this investigation. We outline some of the more interesting directions, in an approximate order of dependency.

20.2.1 Furthering the LSD-SLAM Investigation

Firstly, as we have already alluded to, is to perform an exhaustive design space exploration of LSD-SLAM. This would enable complete validation of the results obtained in this report.

Secondly, we have already in this investigation, defined and characterised the kernels within LSD-SLAM, however this work could be extended by extracting them, into their own self contained bodies of code. The purpose would be to have reusable components,

therefore heading towards a plug-and-play architecture. This would enable modules to shared with other algorithms or out-right replaced, e.g. replacing g2o with Ceres[1]. Possibly before attempting this, it would be worth performing at least one further integration, of a similar SLAM algorithm, so the kernels can compared with another implementation, which would confirm or disprove our decomposition.

20.2.2 Integration and Analysis of More Algorithms

We selected one of a numerous number of non-dense SLAM implementations, and therefore there are many more interesting algorithms (sparse, dense or otherwise) to integrate and investigate.

Sparse SLAM

There are variety of other sparse SLAM algorithms, an interesting one is SVO, Semi-Direct Visual Odometry[2]. SVO is similar to LSD-SLAM in that is summarises the scene into key-frames, and uses intensities to track frames. But, it also uses features, only when initialising new key-frames to define what points to track between the key-frame and new frames.

Dense SLAM

For the sake of completeness, but already mentioned in the SLAMBench paper, is to integrate extensions to KinectFusion, "which scalable in terms of the size of the scene to be reconstructed" [16]. One of their suggestions is to integrate 'Kintinuous' which enables the TSDF volume to 'move'.

20.2.3 Map Comparison

A very important aspect missing from SLAMBench is map comparison. With reference to the report title, we cannot currently answer the question, 'What level of sparsity is good enough?'. Although one can argue that if the SLAM algorithm obtains a good ATE, its internal map must be good. However, this assumption is not suitable when we wish to extract the map, for example to 3D print it. There are three problems which need to be addressed to solve this problem:

1. Select a method to *align* - known as 'registration' - the model and map. This must support sparse maps, frequently represented as point clouds.
2. Determine a *metric* for comparison, ideally this would be a scalar value.
3. Suitable set of *surface models*

[1] Ceres is a library, very similar to g2o, for solving optimisation problems. See `http://ceres-solver.org`.

[2] "SVO: Fast semi-direct monocular visual odometry", by C. Forster, M. Pizzoli, D. Scaramuzza. Published in 'IEEE Intl. Conf. on Robotics and Automation, ICRA', May 2014

Moreover, these must be solved in an automated and deterministic method to satisfy the reproducibility requirement in SLAMBench.

In the literature there are many suggestions at solving parts 1, 2. One solution, 'Registration of Point Cloud Data from a Geometric Optimization Perspective' by Mitra et al, converts the problem so that it can solved by least squares optimisation, which therefore also provides a error metric, the residuals.

Part 3, is the hardest. Of the datasets we have used, only the ICL-NUIM Living Room has a surface model, and as we have discussed some algorithms do not perform well with a synthetic dataset therefore it is not necessarily a fair test. Therefore, some suitable datasets need to be found and/or generated to meet this goal.

Appendices

Appendix A

Additional Result Plots

A.1 KFusion Characterisation

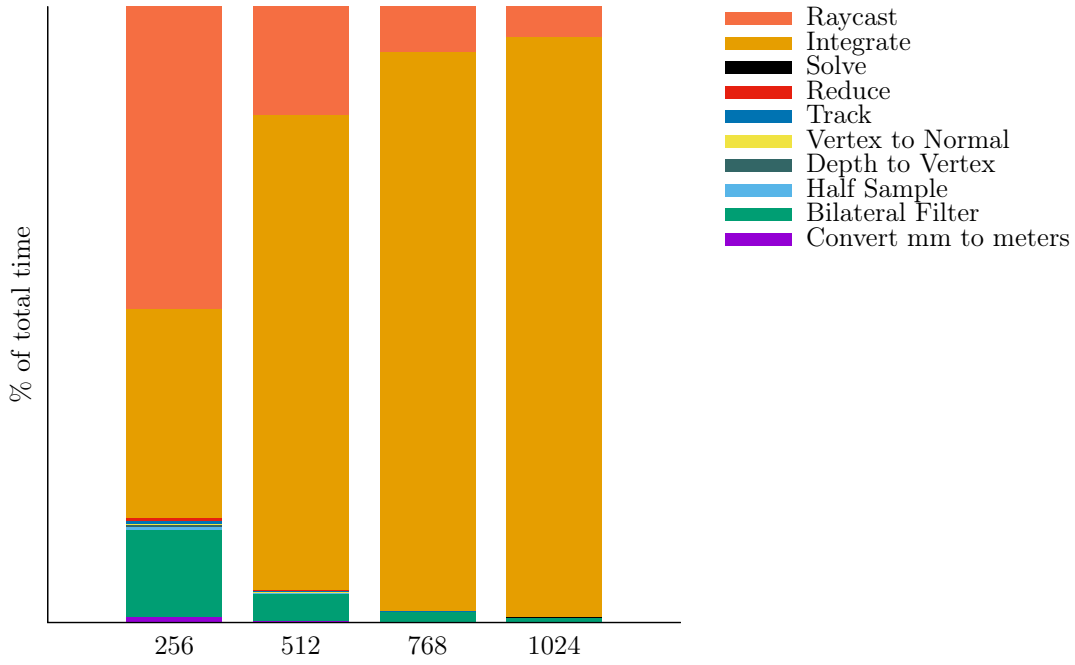


Figure A.1: Kernel timings as a percentage of the total time, in KFusion running on Seyward. We vary the number of voxels (otherwise default parameters) on the TUM RGB-D fr/xyz dataset.

A.2 KFusion Characterisation

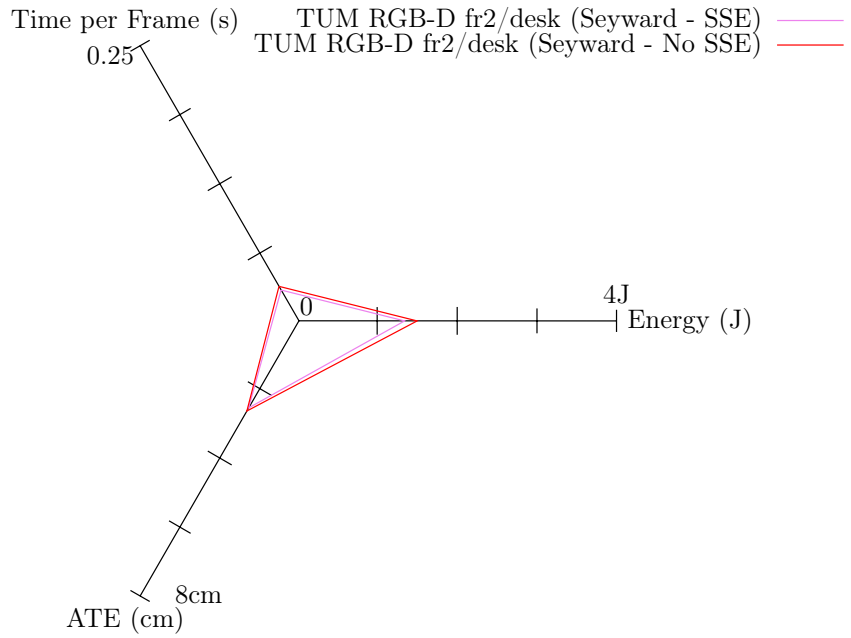


Figure A.2: Comparing metrics on LSD-SLAM with and without SSE optimisations in the tracking kernels, in process-every-frame mode, run on Seyward using TUM RGB-D fr2/desk.

A.3 Design Space Exploraton

A.3.1 Frame Promotion

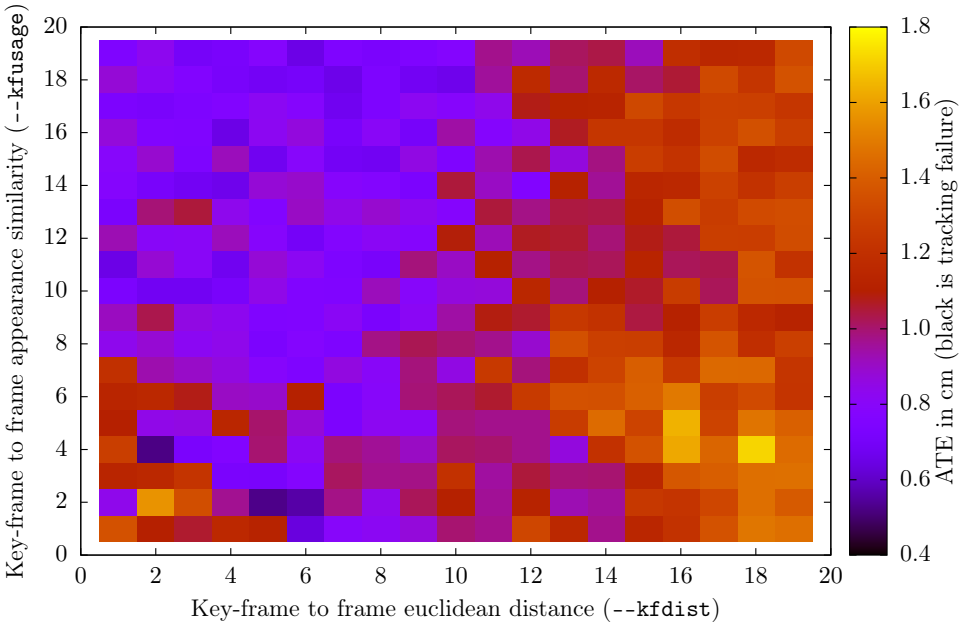


Figure A.3: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/xyz on Seyward with otherwise default parameters. This colours represent ATE (cm).

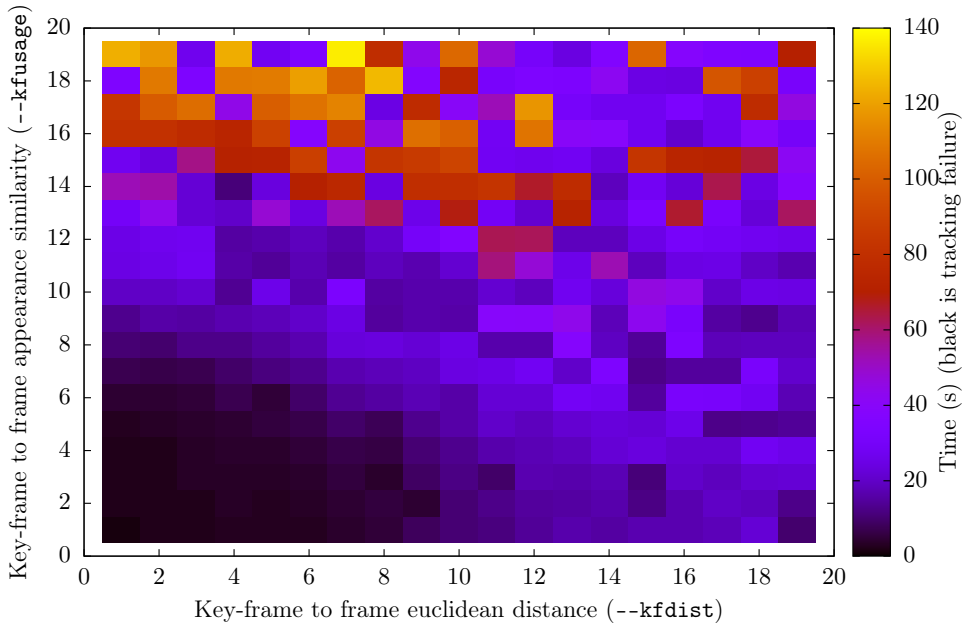


Figure A.4: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/xyz on Seyward with otherwise default parameters. This colours represent finalisation time.

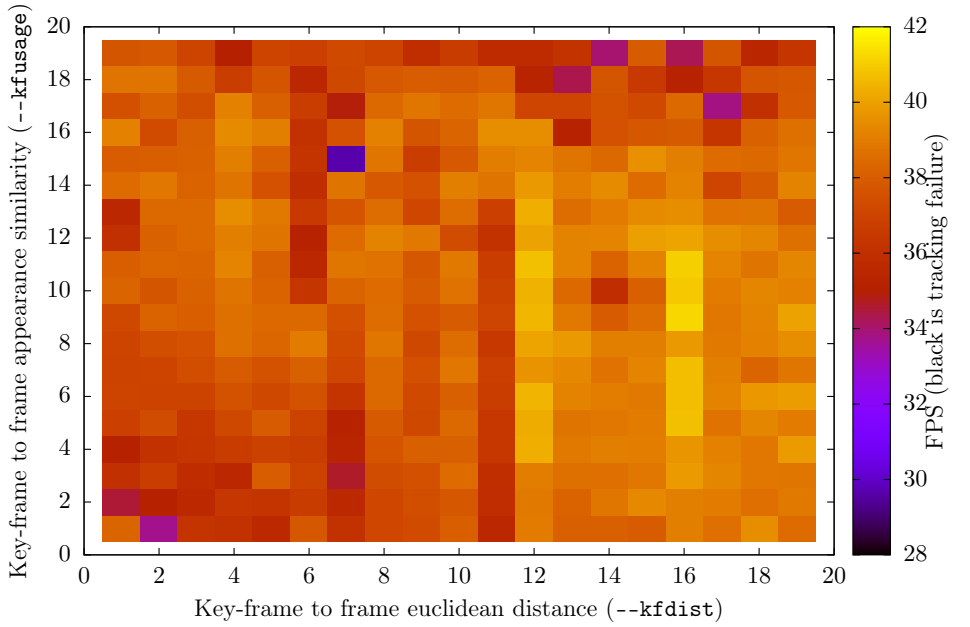


Figure A.5: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using TUM RGB-D fr2/xyz on Seyward with otherwise default parameters. This colours represent FPS.

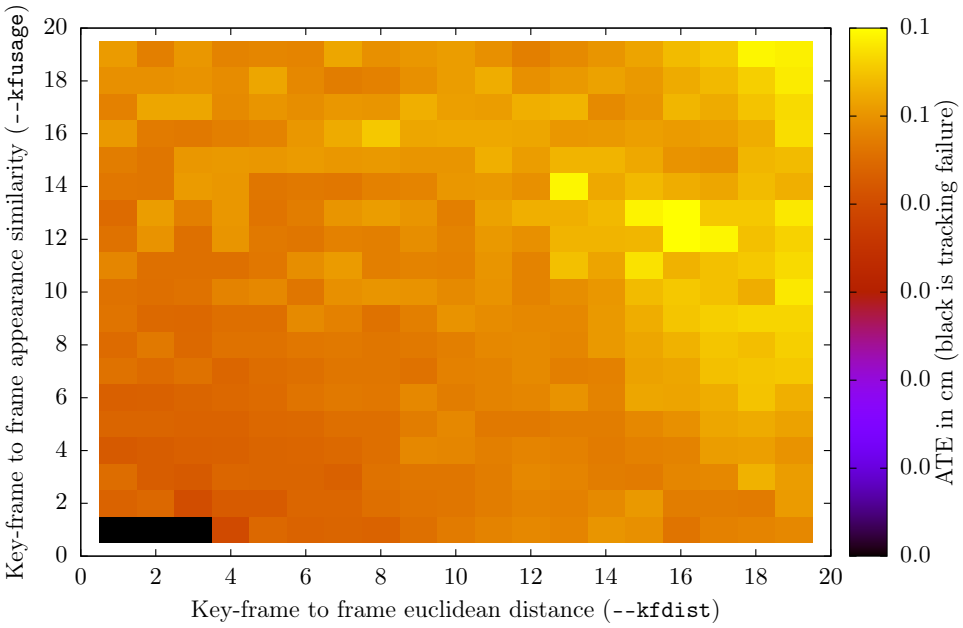


Figure A.6: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using ICL-NUIM Living Room trajectory 2 on Seyward with otherwise default parameters. This colours represent ATE (cm).

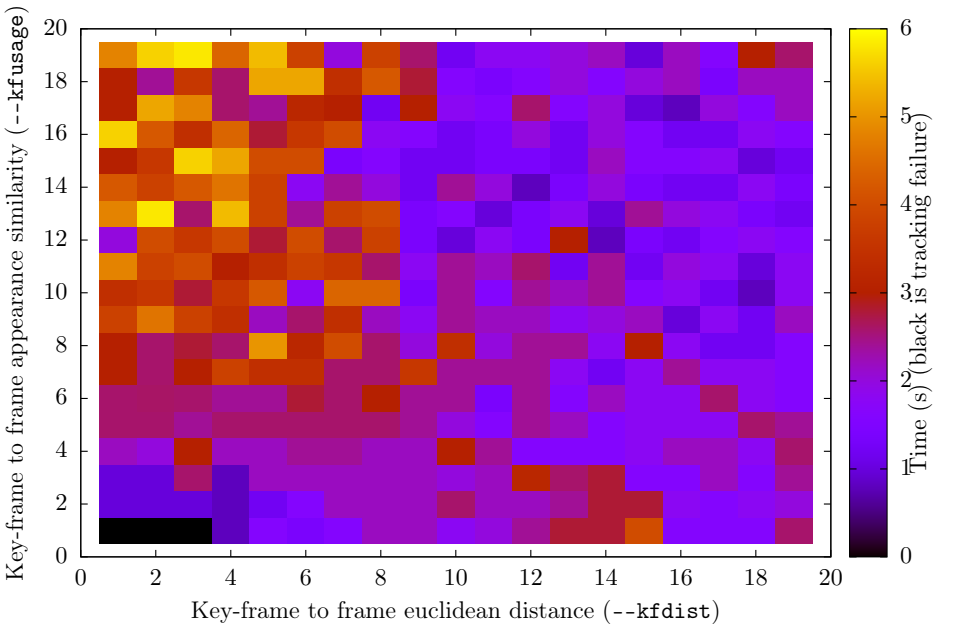


Figure A.7: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using ICL-NUIM Living Room trajectory 2 on Seyward with otherwise default parameters. This colours represent finalisation time.

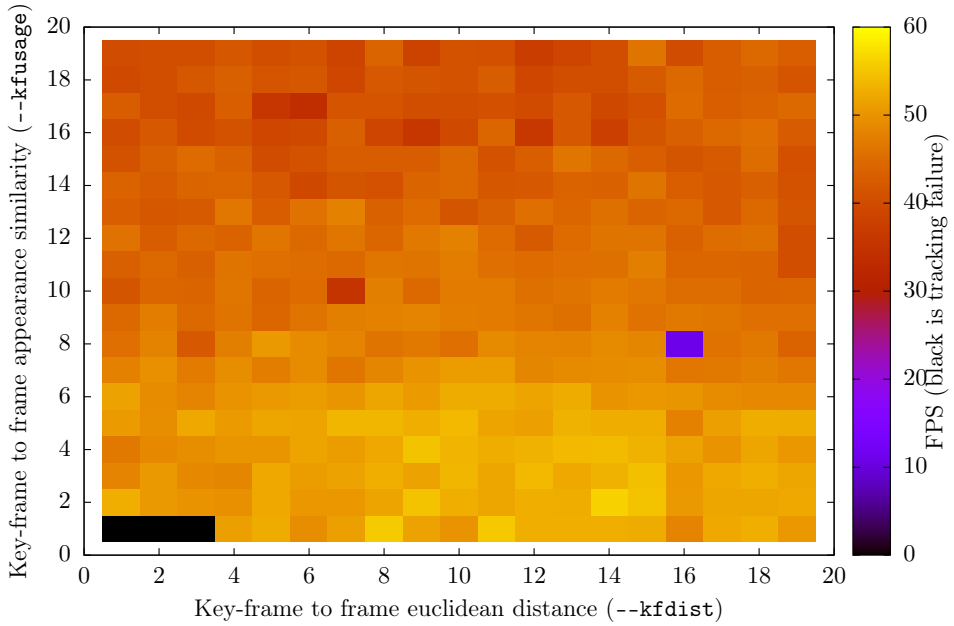


Figure A.8: 2D design space exploration of appearance similarity and distance between frames in LSD-SLAM using ICL-NUIM Living Room trajectory 2 on Seyward with otherwise default parameters. This colours represent FPS.

A.3.2 ODROID Processor Availability

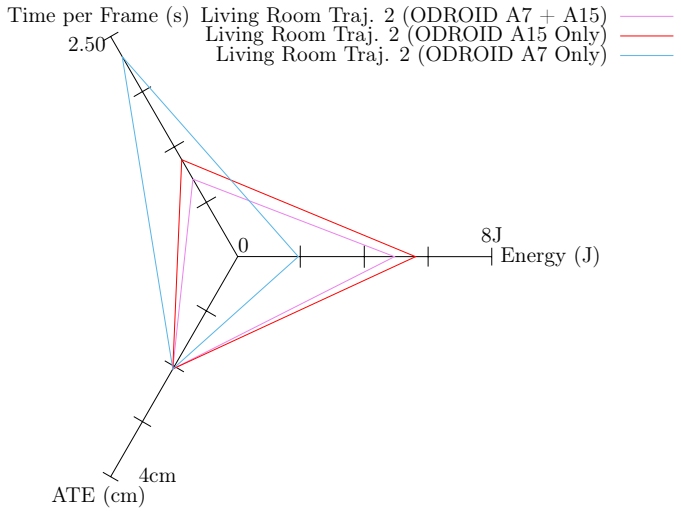


Figure A.9: KFusion running on the ODROID, using default parameters on the ICL-NUIM Living Room trajectory 2. We have varied the availability of processing cores within the big.LITTLE architecture.

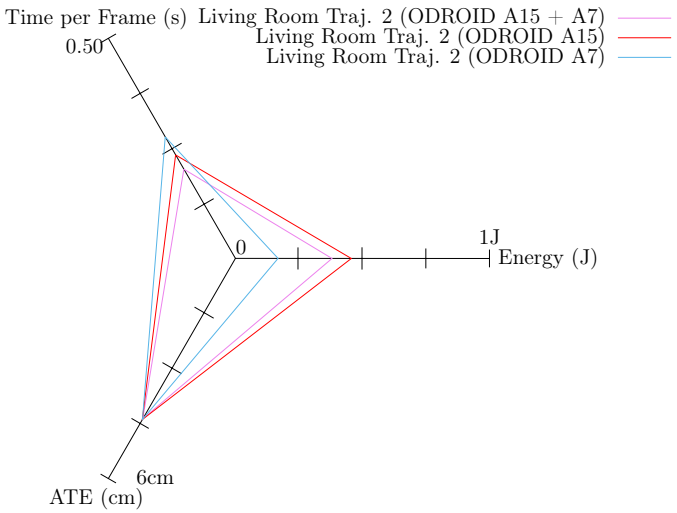


Figure A.10: LSD-SLAM running on the ODROID, using default parameters on the ICL-NUIM Living Room trajectory 2 dataset. We have varied the availability of processing cores within the big.LITTLE architecture.

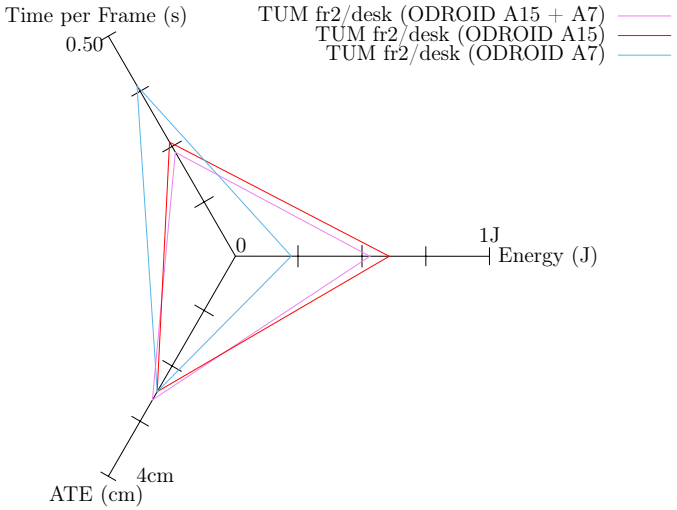


Figure A.11: LSD-SLAM running on the ODROID, using default parameters on the TUM RGB-D fr2/desk dataset. We have varied the availability of processing cores within the big.LITTLE architecture.

Appendix B

Result Reproduction Steps

B.1 Building and Running

To build and run SLAMBench we refer the reader to the appropriate README files contained in the repository.

Implementation	Path to README
KFusion:	`./README`
LSD-SLAM:	`./lsdslam/README`

The repository also contains a sample of the scripts used to perform the tests contained in this report (located in `./scripts/slambench/`).

Appendix C

Hardware and Software Specifications

Here we provide some further hardware and software details.

Seyward

Property	**Data**
Processor	Intel i7-4770 Haswell
CPU cores	4
CPU Frequency	3.4 GHz
Operating System	Ubuntu 14.04LTS
Linux Kernel	3.10.53
Compiler	gcc version 4.8.2 (Ubuntu 4.8.2-19ubuntu1)

ODROID XU3

Property	**Data**
Processor	Exynos 5422
CPU cores	4 (Cortex-A15) + 4 (Cortex-A7)
CPU Frequency	1.8 GHz
Operating System	Ubuntu 14.04LTS
Linux Kernel	3.10.58
Compiler	gcc version 4.8.2 (Ubuntu/Linaro 4.8.2-19ubuntu1)

Glossary[1]

Intensity The degree or amount of some quality ... brightness. [43]

Odometry Determing pose based on sensor readings (The name comes from the odometer sensor [72]). Visual odometry utilises vision to provide the basis for calculating pose [73].

Photometic Comparing the intensities of light from various sources [46].

Pose Comprises its location and orientation relative to a global coordinate frame [72].

Real Time Designating or relating to a system in which input data is processed so quickly so that it is available virtually immediately as feedback to the process from which it emanates, e.g. in a missile guidance system; occurring or available in real time. [74]

[1] These words are defined in the context of SLAM

Bibliography

[1] OED Online. *augmented reality n.* Oxford University Press, December 2014.

[2] Boris Duran and Serge Thill. Rob's robot: Current and future challenges for humanoid robots. In Riadh Zaier, editor, *The Future of Humanoid Robots - Research and Applications*. InTech, January 2012.

[3] National Instruments Corporation. Top Three Challenges in Robotics. `http://www.ni.com/newsletter/50878/en/`. [Online; accessed 26-01-2015].

[4] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Eighteenth National Conference on Artificial Intelligence*, pages 593–598, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[5] Udo Frese. Interview: Is slam solved? *KI - Künstliche Intelligenz*, 24(3):255–257, 2010.

[6] Thomas Lemaire, Cyrille Berger, Il kyun Jung, and Simon Lacroix. Vision-based slam: Stereo and monocular approaches. Technical report, Int. J. Compt. Vision, 2006.

[7] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localisation and mapping (slam): Part i the essential algorithms. *IEEE Robotics & Automation Magazine*, 2:2006, 2006.

[8] Zhaoyang Lv. Visual SLAM: a tour from sparse to dense. `http://www.cc.gatech.edu/~afb/classes/CS7495-Fall2014/presentations/visual_slam.pdf`. [Online; accessed 09-01-2015].

[9] Richard Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew Davison, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and Augmented Reality (ISMAR), 2011 10th IEEE International Symposium on*, UIST '11, pages 127 – 136. IEEE, 2011.

[10] Andrew J. Davison, Ian D. Reid, Nicholas D. Molton, and Olivier Stasse. Monoslam: Real-time single camera slam. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(6):1052–1067, June 2007.

[11] Georg Klein and David Murray. Parallel tracking and mapping for small AR workspaces. In *Proc. Sixth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'07)*, Nara, Japan, November 2007.

[12] Richard A. Newcombe, Steven J. Lovegrove, and Andrew J. Davison. Dtam: Dense tracking and mapping in real-time. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2320–2327, Washington, DC, USA, 2011. IEEE Computer Society.

[13] cogrob/ENSTA/JCB. Loop Closure Detection. `http://cogrob.ensta-paristech.fr/loopclosure.html`. [Online; accessed 05-06-2015].

[14] T. Schöps, J. Engel, and D. Cremers. Semi-dense visual odometry for AR on a smartphone. In *International Symposium on Mixed and Augmented Reality*, September 2014.

[15] J. Engel, T. Schöps, and D. Cremers. LSD-SLAM: Large-scale direct monocular SLAM. In *eccv*, September 2014.

[16] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham D. Riley, Nigel Topham, and Steve Furber. Introducing slambench, a performance and accuracy benchmarking methodology for SLAM. *CoRR*, abs/1410.2167, 2014.

[17] Engineering and Physical Sciences Research Council EPSRC. PAMELA: a Panoramic Approach to the Many-CorE LAndsape - from end-user to end-device: a holistic game-changing approach. `http://gow.epsrc.ac.uk/NGBOViewGrant.aspx?GrantRef=EP/K008730/1`. [Online; accessed 06-01-2015].

[18] PAMELA Project. About. `http://apt.cs.manchester.ac.uk/projects/PAMELA/about/index.html`. [Online; accessed 29-01-2015].

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[20] Prof. Michael Hicks. Types of Parallelism. `http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concurrency-basics.pdf`. [Online; accessed 05-06-2015].

[21] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[22] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, September 1972.

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

[24] Jonathan Christensen and StackOverFlow. Least squares - Mean absolute error OR root mean squared error? - Cross Validated. `http://stats.stackexchange.com/questions/48267/mean-absolute-error-or-root-mean-squared-error`. [Online; accessed 12-05-2015].

[25] Joseph A. Paradiso and Thad Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, January 2005.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[27] Microsoft. DepthImageFormat Enumeration. `http://msdn.microsoft.com/en-us/library/microsoft.kinect.depthimageformat.aspx`. [Online; accessed 07-01-2015].

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

[29] Leow Wee Kheng. Camera Models and Imaging. `http://www.comp.nus.edu.sg/~cs4243/lecture/camera.pdf`. [Online; accessed 06-06-2015].

[30] Hauke Strasdat. *Local Accuracy and Global Consistency for Efficient Visual SLAM*. PhD thesis, Department of Computing, Imperial College London, October 2012.

[31] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, and Andrew Fitzgibbon. Kinectfusion: Real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 559–568, New York, NY, USA, 2011. ACM.

[32] Edwin Chong and Stanislaw Zak. *An Introduction to Optimization*. Wiley, 4 edition, 2013.

[33] Sam Roweis. Levenberg-Marquardt Optimization. `https://www.cs.nyu.edu/~roweis/notes/lm.pdf`. [Online; accessed 12-05-2015].

[34] Microsoft. Xbox Kinect Motion Sensors - Games, Photos & News | Xbox.com UK. `http://www.xbox.com/en-GB/Kinect`. [Online; accessed 06-01-2015].

[35] Daniel Korcz. Volumetric Range Image Integration. `http://www.ifp.uni-stuttgart.de/lehre/diplomarbeiten/korcz/`. [Online; accessed 07-01-2015].

[36] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 839–, Washington, DC, USA, 1998. IEEE Computer Society.

[37] Szymon Rusinkiewicz and Marc Levoy. Efficient variants of the ICP algorithm. In *Third International Conference on 3D Digital Imaging and Modeling (3DIM)*, jun 2001.

[38] Brian Peasley and Stan Birchfield. Replacing projective data association with lucas-kanade for kinectfusion. In *ICRA*, pages 638–645. IEEE, 2013.

[39] Kok lim Low. Linear least-squares optimization for point-to-plane icp surface registration. Technical report, University of North Carolina at Chapel Hill, 2004.

[40] Microsoft. Kinect for Windows SDK. `http://msdn.microsoft.com/en-us/library/hh855347.aspx`, 30-09-2013. [Online; accessed 02-01-2015].

[41] Gerhard Reitmayr TU Graz. KFusion. `https://github.com/GerhardR/kfusion`, 01-04-2013. [Online; accessed 06-01-2015].

[42] pointclouds.org. Documentation - Point Cloud Library. `http://pointclouds.org/documentation/tutorials/using_kinfu_large_scale.php`. [Online; accessed 06-01-2015].

[43] OED Online. *intensity, n.* Oxford University Press, December 2014.

[44] David Jacobs. Image Gradients. `http://www.cs.umd.edu/~djacobs/CMSC426/ImageGradients.pdf`. [Online; accessed 10-04-2015].

[45] José-Luis Blanco. A tutorial on se(3) transformation parameterizations and on-manifold optimization. Technical report, University of Malaga, September 2010.

[46] OED Online. *photometer, n.* Oxford University Press, December 2014.

[47] J. Engel, J. Sturm, and D. Cremers. Semi-dense visual odometry for a monocular camera. In *IEEE International Conference on Computer Vision (ICCV)*, Sydney, Australia, December 2013.

[48] Stack Exchange and Respective authors. What is regularization in plain english? `http://stats.stackexchange.com/questions/4961/what-is-regularization-in-plain-english`. [Online; accessed 09-06-2015].

[49] TUM Computer Vision Group. Computer Vision Group - Useful tools for the RGB-D benchmark. `https://vision.in.tum.de/data/datasets/rgbd-dataset/tools`. [Online; accessed 05-05-2015].

[50] TUM Computer Vision Group. `tum-vision/lsd_slam`. `https://github.com/tum-vision/lsd_slam`. [Online; accessed 21-01-2015].

[51] MIT / Open Source Initiative. The MIT License (MIT) | Open Source Initiative. `http://opensource.org/licenses/MIT`. [Online; accessed 09-04-2015].

[52] Open Source Initiative. Frequently Answered Questions | Open Source Initiative. `http://opensource.org/faq`. [Online; accessed 09-04-2015].

[53] GNU. What is Copyleft? - GNU Project - Free Software Foundation. `https://www.gnu.org/copyleft/`. [Online; accessed 09-04-2015].

[54] Opensource Robotics Foundation. ROS.org | Powering the world's robots. `http://www.ros.org/`. [Online; accessed 03-04-2015].

[55] Opensource Robotics Foundation. ROS/Installation - ROS Wiki. `http://wiki.ros.org/ROS/Installation`. [Online; accessed 08-04-2015].

[56] Thomas Wheelan / Github.com. mp3guy/lsd_slam. `https://github.com/mp3guy/lsd_slam`. [Online; accessed 08-04-2015].

[57] Hauke Strasdat. strasdat/Sophus. `https://github.com/strasdat/Sophus`. [Online; accessed 06-06-2015].

[58] Eigen Library Authors. Eigen. `http://eigen.tuxfamily.org/index.php?title=Main_Page`. [Online; accessed 06-06-2015].

[59] Edward Rosten. TooN: Tom's Object-oriented numerics library. `http://www.edwardrosten.com/cvd/toon.html`. [Online; accessed 06-06-2015].

[60] C. Kerl. Odometry from rgb-d cameras for autonomous quadrocopters. Master's thesis, Technical University Munich, Germany, Nov. 2012.

[61] Jakob Engel. Computer Vision Group - Jakob Engel. `http://vision.in.tum.de/members/engelj`. [Online; accessed 05-05-2015].

[62] A. Handa, T. Whelan, J.B. McDonald, and A.J. Davison. A benchmark for RGB-D visual odometry, 3D reconstruction and SLAM. In *IEEE Intl. Conf. on Robotics and Automation, ICRA*, Hong Kong, China, May 2014. (to appear).

[63] TUM Computer Vision Group. Computer Vision Group - File Formats. `http://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats`. [Online; accessed 27-05-2015].

[64] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf`. [Online; accessed 05-05-2015].

[65] T. Whelan, M. Kaess, M.F. Fallon, H. Johannsson, J.J. Leonard, and J.B. McDonald. Kintinuous: Spatially extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, Sydney, Australia, Jul 2012.

[66] Ming Zeng, Fukai Zhao, Jiaxiang Zheng, and Xinguo Liu. A memory-efficient kinectfusion using octree. In *Proceedings of the First International Conference on Computational Visual Media*, CVM'12, pages 234–241, Berlin, Heidelberg, 2012. Springer-Verlag.

[67] Niessner, Matthias and Zollhöfer, Michael and Izadi, Shahram and Stamminger, Marc. Real-time 3d reconstruction at scale using voxel hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11, nov 2013.

[68] Microsoft. Kinect Sensor. `http://msdn.microsoft.com/en-gb/library/hh438998.aspx`. [Online; accessed 12-01-2015].

[69] John MacCormick. How does the Kinect work? `http://www.cs.bham.ac.uk/~vvk201/Teach/Graphics/kinect.pdf`. [Online; accessed 01-05-2015].

[70] Microsoft. Lighting for Kinect | Examples of Good and Bad Lighting for Kinect. `http://support.xbox.com/en-US/xbox-360/kinect/lighting`. [Online; accessed 29-05-2015].

[71] Rainer Kummerle, G. Grisetti, H. Strasdat, Kurt Konolige, and Wolfram Burgard. g2o: A general framework for graph optimization. In *ICRA*, Shanghai, 05/2011 2011.

[72] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005.

[73] Wikipedia Authors. Visual odometry - Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Visual_odometry`. [Online; accessed 13-06-2015].

[74] OED Online. *real time, adj.* Oxford University Press, December 2014.