KudanSLAM

Technical Information

Contents

Introduction

Simultaneous Localisation and Mapping (SLAM) has become a highly important topic within the computer vision community, and is receiving particular interest from a large number of industries. With a variety of SLAM systems being made available, from both academia and industry, it is worth exploring what exactly is meant by SLAM. This introduction will give a brief overview of some of the fundamental concepts that make up KudanSLAM.

SLAM

Simultaneous Localisation and Mapping (SLAM) has a long history and a diversity of techniques has been adopted to solve it at different times. In this article we focus on the current mechanism of SLAM as implemented in KudanSLAM. SLAM, by definition, needs to solve two problems at the same time:

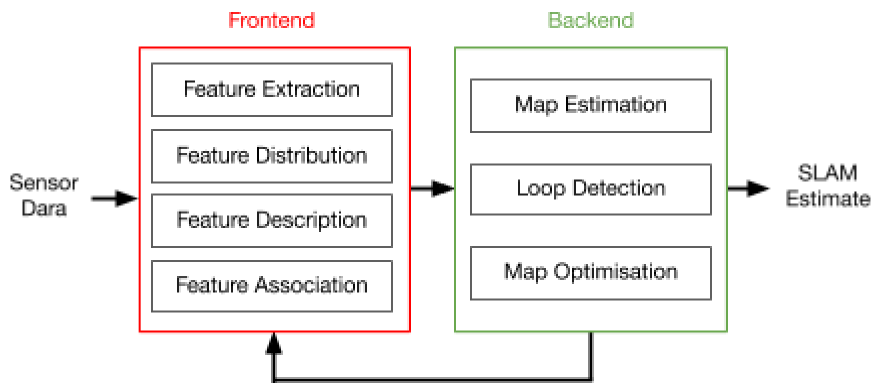
- Localising a sensing agent in an environment.
- Mapping of the environment itself.

It thus stands to reason that we can split the problem in two parts: a tracking mechanism solves the localisation part by comparing new input data to the currently existing map, and a mapping part that supports the tracking system by providing, maintaining and expanding the map itself based on the information that comes from the input data sensed by the tracking part.

KudanSLAM is primarily a Visual SLAM system, which means that the main input data is a camera feed. This feed can be optionally augmented by a variety of sensors:

- A second camera to create a Stereo Visual SLAM system.
- A depth sensor in the case of RGBD SLAM.
- An Inertial Measurement Unit (IMU) for Visual Inertial SLAM.

KudanSLAM divides tracking and mapping tasks in two main threads, that work in parallel to provide the real time position and orientation of the main camera and an optimised map of the environment.

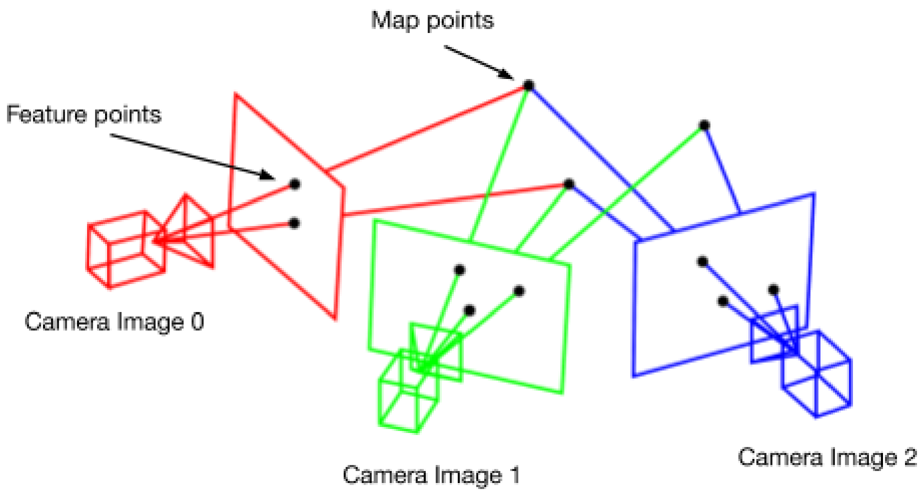


Tracking

The tracking thread provides real time localisation by extracting relevant information from the main camera image and from the auxiliary information if available, then comparing the extracted information with the existing map structure. The KudanSLAM map structure is made of triangulated 3D map points created by tracking feature points over a number of frames or between stereo frames. During tracking each input frame the image is analyzed to extract feature points, which are distinctive points that are selected as they will be re-detectable in future images. In KudanSLAM corner shapes are used for this purpose at various scale levels using the FAST algorithm.

If the current map of the environment is empty there is nothing for the tracker to track against, this is solved using an initialisation procedure. The initialisation procedure is different depending on the available sensors, for example a Stereo Visual SLAM system can create an initial map by triangulating matching feature points between the stereo images. The result of the initialisation procedure is a map containing some a number of 3D map points which were detected in a number of frames. When frames are used to expand the create or expand the map area we call these frames keyframes.

Once there is a map to track against the tracking procedure will use this to compute an accurate drift free position and orientation of the camera. Tracking against the map is achieved by matching feature points from new images to the previously mapped features in the map. Depending on the amount of movement of the camera, the tracking thread might decide to ask the mapping thread to expand the map to include the current location by creating a new keyframe.



Mapping

The mapping thread needs to provide many different features to support tracking. Mapper functionalities include providing existing keyframe information to the tracker, creating new keyframes when asked by the tracker, optimising and refining the map data, supporting relocalisation, and detecting loops. Many of those tasks are performed in parallel in KudanSLAM taking advantage of the multithreading capabilities of today's processors.

Since SLAM is a is real time application, the most urgent mapping operations are not applied to the full map but only to the relevant local part. The local part of the map is composed of the most recently created keyframe, and the set of keyframes that are close to it in terms of co-visibility. This allows the mapper to quickly update information for the tracker. As new keyframes are created KudanSLAM maintains a fast runtime by connecting new keyframes only to the parts of the map that are co-visible to them. By carefully controlling connections between keyframes map optimisation, creation of new map points, and culling of redundant information can be performed often and quickly.

To enable relocalisation and loop closure functionality, the mapper also maintains a visual vocabulary and recognition database of all the keyframes in the map. This information allows for fast relocalisation anywhere in the map should tracking loss occur, and allows a background mapping process to detect visual loops. By detecting visual loops in the map and performing loop closure it is possible to correct drift in the trajectory.

When possible, global optimisation is performed in order to propagate local corrections through the map. This global optimisation gets inevitably more slow as the map grows, so it’s done only occasionally in a separate thread when the computational resources allow for it. Global optimisation can also be triggered by detected visual loops, in order to propagate drift correction to the surrounding areas.

Relocalisation

During visual tracking it is possible for the camera to be occluded or rotated too suddenly for the new camera frame to be tracked against the existing map. This causes tracking to be lost. In this situation there are two options which can be used to recover depending on the use case of KudanSLAM: The map could be cleared and tracking re-initialised, or, more commonly, the system can attempt to relocalise the camera against the map.

During relocalisation, the new camera frames are compared visually with a database of the full map to find visually similar keyframes. If any visually similar keyframes are found they are used as candidates for resuming tracking. If a candidate keyframe for relocalisation is found, tracking is tentatively resumed from its pose, and for the following few incoming frames the tracking quality is thoroughly checked to ensure the candidate was the correct one. If the checks are successful, KudanSLAM continues tracking and mapping as normal, otherwise it will continue trying to relocalise against the map. During this trial period, just after relocalisation, the map is not expanded in order to avoid potential corruption.

Approach

At Kudan, our approach to software development is based on several key principles which ensures that we remain ahead in a rapidly developing and competitive field. These principles are based on providing highly-performant cross platform SLAM software and a supportive base for our clients to ensure they are able to receive all the advantages of using our KudanSLAM system. Moreover, our customer-based approach allows us to deliver a boutique, tailor-made SLAM software package, allowing clients to fully exploit the capabilities of their hardware.

A widely used academic SLAM system which is used extensively in the field is ORB-SLAM2, an open-source, freely available system. This section will describe several key areas in which KudanSLAM offers obvious competitive advantages in terms of performance and functionality when compared to ORB-SLAM2.

Continuous Improvement

One of our key principles is the continuous improvement of our software. We consider the lack of development of ORB-SLAM2 to be a major limitation. The development of ORB-SLAM2 was discontinued in 2017, when the academic paper was published, which is not an uncommon situation with academic software. At Kudan, our software is being continually improved and tested and over the years this can be seen in the large performance gains we have achieved. These performance gains have been realised through careful consideration of the SLAM problem to remove bottlenecks and through platform specific optimisations.

Platform Optimisation

Another key principle at Kudan is writing software that is optimised for the hardware it is to be executed on. This principle is delivered through the use of custom-designed algorithms which make the use of more processor specific instruction sets such as AVX2 and NEON. See the next section for the complete list of optimisations supported by KudanSLAM. This approach results in superior performance when used on dedicated hardware platforms.

In addition, KudanSLAM is designed to work across a multitude of operating systems, a major advantage of using KudanSLAM compared to our rivals. The combined use of processor specific languages and our cross-platform approach makes KudanSLAM a highly versatile software platform that can be used across a wide range of hardware types.

Support

As the engineers at Kudan are computer vision experts they are able to support our clients use of KudanSLAM. This allows users of KudanSLAM to get the most out of our software and maximise the performance of their products. With academic software there is rarely any form of support, and additionally at Kudan we dive deep into customer requirements to ensure we fully understand what they want to achieve. As we are subject matter experts we can often find ways to support our clients above and beyond their original ideas.

Designed for the Real-World

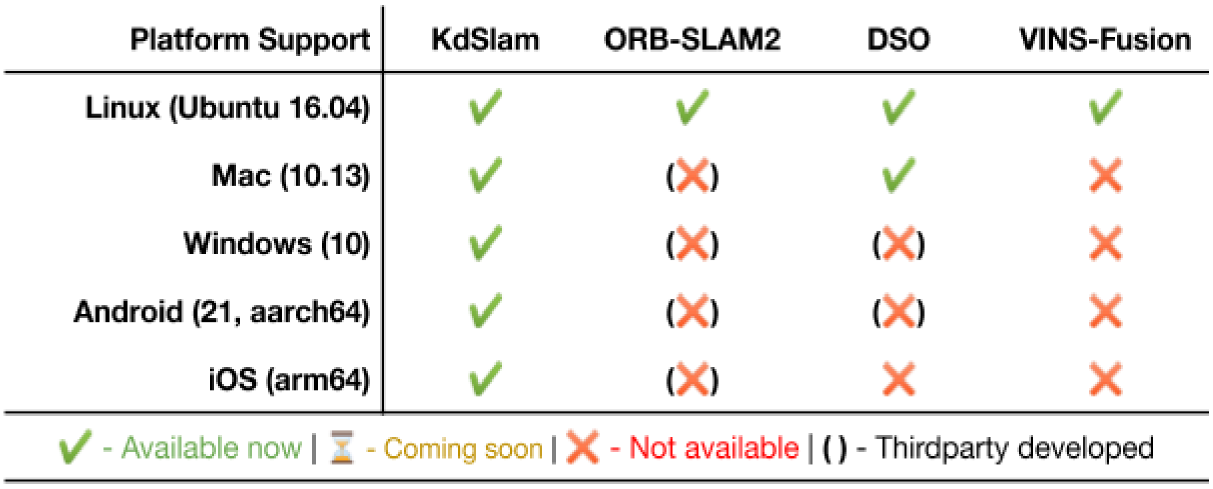
At Kudan, we pride ourselves upon taking an original approach when developing the key components of our software. Rather than designing software for simulation and theoretical outcomes, we design our software starting from real-world use cases. KudanSLAM is developed and tested using data obtained from several different sources aiming to test our algorithms' performance in as many environments and use cases as possible in order to guarantee reliable performance and robustness to challenging conditions. All our code is thoroughly tested for bugs, then performance evaluated in a battery of tests that measure our algorithms against various datasets, covering both the ones most used in academic literature, and bespoke ones that are designed to test more realistic and difficult conditions. In addition, synthetic data is also designed as input for our tests in order to specifically stress-test some core aspects of the API components. Our engineers also perform different rounds of manual testing on the finished KudanSLAM APIs, using it with live cameras, different sensor configurations and in different environments to catch any possible problem before the APIs are released. The performance of different algorithms is analysed by our experts and used to create highly-optimised algorithms which ultimately results in a more performant pipeline in the real-world. Our approach produces algorithms which are faster, more robust, and more accurate when compared to academic and other custom solutions.

Hardware and Software

The previous section highlighted the high-level strategies at Kudan, namely the key principles that are adopted at Kudan to deliver highly performant software with a client-based approach. To further elaborate on the benefits of KudanSLAM over other SLAM systems that are available, this section will focus on the low-level, and more technical aspects of how the software is optimised. In addition to the development side, Kudan offers superlative support to its clients, working with them to ensure they obtain the best performance from our SLAM system on their desired platform and hardware configurations.

Operating Systems and Optimisation

One of the underlying philosophies behind the development of KudanSLAM is to produce software that is versatile, running on a multitude of platforms. This versatility is highlighted in table 1 below, KudanSLAM is supported on all major OS's, including mobile platforms.



Platform Support	KdSlam	ORB-SLAM2	DSO	VINS-Fusion
Linux (Ubuntu 16.04)	✓	✓	✓	✓
Mac (10.13)	✓	(✗)	✓	✗
Windows (10)	✓	(✗)	(✗)	✗
Android (21, aarch64)	✓	(✗)	(✗)	✗
iOS (arm64)	✓	(✗)	✗	✗

✓ - Available now | ⏳ - Coming soon | ✗ - Not available | ( ) - Thirdparty developed

Table 1 - Operating systems supported by KudanSLAM

In addition, KudanSLAM also supports processor specific languages (See Table 2) such as Avx2 and ARM Neon , allowing KudanSLAM to run on numerous hardware/software combinations to ensure optimised performance on these platforms. This is an obvious advantage of KudanSLAM over other academic SLAM systems, which primarily target only Linux and are not optimised. Furthermore, developers at Kudan are currently working on increasing the range of devices KudanSLAM will operate on, with future additions taking advantage of Graphical Processor Units (GPUs), Field Programmable Gate Arrays (FPGAs) and Digital Signal Processors (DSPs). This will further increase the scope of KudanSLAM, making it one of the most versatile SLAM systems on the market.

Optimisation	KdSlam	ORB-SLAM2	DSO	VINS-Fusion
Unoptimised C++	✔	✔	✔	✔
x86 AVX2	✔	✗	✗	✗
ARM NEON	✔	✗	✗	✗
FPGA	✓	✗	✗	✗
GPU	⏳	✗	✗	✗
DSP	✓	✗	✗	✗

✔ - Available | ✓ - Alpha | ⏳ - Coming soon | ✗ - Not available | ( ) - Thirdparty

Table 2 - Architecture specific optimisations supported by KudanSLAM

Sensor Data

The versatility of KudanSLAM also extends to its support for sensors, with a wide range of hardware sensors supported as shown in Table 3 below. Both monocular and stereo cameras are supported in combination with other sensors such as IMU devices offering superior tracking. Stereo cameras supported by KudanSLAM include LeadSense, HIK, RealSense and Leopard cameras, with depth cameras also supported.

Sensors	KdSlam	ORB-SLAM2	DSO	VINS-Fusion
Monocular Camera	✔	✔	✔	✔
Stereo Camera	✔	✔	(✗)	✔
RGBD	✔	✔	✗	✗
Fisheye Cameras	✓	(✗)	(✗)	✔
IMU Support	✔	✗	(✗)	✔
GPS Support	⏳	✗	✗	✔
LiDAR Support	✓	✗	✗	✗

✔ - Available | ✓ - Alpha | ⏳ - Coming soon | ✗ - Not available | ( ) - Thirdparty

Table 3 - Sensors supported by KudanSLAM

Hardware

At Kudan, we produce custom-made, bespoke camera rigs for customers using lenses and machine vision camera from our trusted suppliers. However, we are also flexible to client needs, and, if required, are able to source hardware from other sources or use the customers own sensors. Examples of lens/camera rigs we have used KudanSLAM with for clients in the past include:

- Leadsense, a stereo camera and IMU in a single hardware synchronised unit.
- Leopard, a stereo camera with a small baseline and fixed lenses.
- HIK cameras with SpaceCom or Theia lenses, either as a monocular camera or a stereo unit with any baseline, synchronised with additional hardware.
- Outdoor cameras: GoPro Hero5
- Phones: Pixel2, Iphone7
- Tablets: Ipad
- Other boutique stereo cameras: Perceptin, Duo, Inuitive, Realsense, MyntEye, and others.

In addition, we have tested with rigs combining cameras and IMU devices such as the EuRoC MEMS IMU (ADIS16448 @ 200 Hz) allowing for visual inertial SLAM when used in conjunction with our software.

Features and API

As a continuously developed system KudanSLAM has a large number of advanced features that have been carefully integrated to cooperate efficiently and effectively.

Features

In table 4, below, we highlight some of the features available in KudanSLAM as compared to the limited features in academic systems. This is to be expected with academic slam systems as they are developed over a short period of time to highlight a small novel discovery or feature, which is subsequently published.

Features	KdSlam	ORB-SLAM2	DSO	VINS-Fusion
Input Masking	✔	(✗)	✗	✗
Point Features	✔	✔	✗	✔
Edge Features	⌛	(✗)	✗	✗
Photometric Features	✗	✗	✔	✗
Odometry Mode	✔	✗	✔	✗
Relocalisation	✔	✔	✗	✔
Cross-Camera Support	✔	✗	✗	✗
Loop Closure	✔	✔	(✗)	✔
Multi-Layer Mapping	✓	(✗)	✗	✗
Map Splitting / Merging	✓	✗	✗	✗
Local Map Optimisation	✔	✔	✔	✔
Global Map Optimisation	✔	✔	✗	✔
Map Saving and Loading	✔	(✗)	✗	✗
ROS Integration	✓	✔	✔	✔

✔ - Available | ✓ - Alpha | ⌛ - Coming soon | ✗ - Not available | ( ) - Thirdparty

Table 4 - SLAM features supported by KudanSLAM

Sample API Usage

The design of the KudanSLAM API allows powerful features to be accessed via a simple user interface. This is illustrated in the example code below which initialises and runs SLAM using a live camera:

```
// Create slam object
KdSlam slam;
// Configure slam settings
slam.loadCameraCalibration(calibrationFile);
slam.loadVocabulary(vocabFile);
slam.setExpectedFps(slamFps);
// Initialise KudanSLAM
slam.initialise();
// Optionally load a map
slam.loadSlamMap(mapFileInput);

while(running) {
    // Get sensor input
    auto currentFrame = getFrameFromCamera();
    // Processing
    slam.processFrame(currentFrame);
    // Output
    auto cameraTranslation = slam.getCameraTranslation();
    auto cameraRotation = slam.getCameraRotation();
    auto projectedPoints = slam.getMapPoints2D();
    auto mapPoints = slam.getMapPoints3D();
    auto keyframePoses = slam.getKeyframePoses();
    auto cameraTrajectory = slam.getCameraTrajectory();
}

// Optionally save the map
slam.saveSlamMap(mapFileOutput);
```

Below is a further break down of each function in the example code and what that particular function does:

slam.loadCameraCalibartion(calibrationFile)
Loads the calibration file for the desired sensor using the designated file path supplied by the user.

slam.loadVocabulary(vocabFile)
Loads a vocabulary file from disc using the designated file path supplied by the user.

slam.setExpectedFps(slamFps)
Sets SLAM to try and target the requested FPS. Whether the desired FPS can be achieved will be dependent on the performance characteristics of the hardware KudanSLAM is being executed on.

slam.initialise()
Initialises the SLAM system. It is important that the loading and parameter function calls above are conducted before calling this function.

slam.loadSlamMap(mapFileInput)
Loads a slam map from disc using the designated file-path. This map would have been saved to file from a previous run.

slam.processFrame(currentFrame)
This function is what actually processes the current frame and produces the internal output. The currentFrame can be obtained from a live camera or file.

slam.getCameraTranslation()

This function will return a vector containing the computed camera translation after running slam on the frame.

slam.getCameraRotation()

Calling this function will return a 3x3 matrix containing the computed camera rotation after running slam on the current frame.

slam.getMapPoints2D()

After processing SLAM, a list of 2D map points can be retrieved by calling this function.

slam.getMapPoints3D()
After processing SLAM, a list of 3D map points can be retrieved by calling this function.

slam.getKeyframePoses()

After processing SLAM, a list of keyframe poses can be obtained by calling this function.

slam.getCameraTrajectory()

The current camera trajectory can be determined through a call to this function which will return a 4x4 matrix.

slam.saveSlamMap(mapFileOutput);
The SLAM map can be saved to disc using the user-defined file-path. This allows the user to load the map back into the SLAM system as described in the *loadSlamMap()* function.

Performance

At Kudan, we have been developing SLAM algorithms that can be deployed on a variety of different products that use different sensors, such as drones, robots, vehicles, and mobile devices. Continually evaluating the performance of our SLAM system and discovering limitations is an important part of our development process, to ensure we keep pushing the boundaries of what is possible.

How to Evaluate SLAM Accuracy

In order to evaluate a SLAM system accurately we first require accurate ground truth data regarding the pose and trajectory of the camera. This may seem simple to obtain at first, but it is difficult to acquire accurately in practice as we will explain in this section. Here are a few techniques we deploy for evaluating SLAM systems at Kudan.

CG Rendering

This involves generating a synthetic scene on a computer software to test our SLAM system virtually. Using this method, we have precise control of all the environment and lightning conditions, and we know the position/orientation and trajectory of the camera with absolute certainty as it is specified by us in the first place, so we can calculate the accuracy of the SLAM system inside the simulation with total precision. However, as the scene is computer-generated, it does not account for many real-world noise sources, and like higher-order camera lens artifacts and various sources of inaccuracy in the sensor readings. This means that virtual scenes alone are not an accurate indicator of real-world performance. Instead, this method allows us to quickly test our algorithm against precisely-designed virtual scenarios and identify fundamental issues before we introduce real-world conditions.

Black Room

As implied by the name, this method includes using a physical rig inside a dark room to reduce noise from unaccounted light sources and reflections. The setup requires the experimenter to perform a variety of different tests and physical movements, such as sliding the camera across a straight rail or rotating it on a turntable to simulate cyclic motion. As we have the dimensions of the rig and the time taken to move the camera, we can extract the position and orientation of the camera at different points in time. However, as the experimenter relies on experience to move the camera and the movement is not externally tracked, there is room for error which makes our ground truth an approximation at best. This type of setup is mostly useful for conducting small-scope tests and evaluate the viability of the SLAM system outside of a simulation. With respect to CG rendering, it introduces real cameras and movements in the testing, at the cost of reduced faithfulness of the ground truth but maintaining strict control on the environment and lightning conditions.

HTC Vive

This setup includes the use of a HTC Vive tracker connected to a camera rig which can be moved around within the HTC Vive ‘play area’. We have developed custom code which allows us to continuously track the orientation and trajectory of the camera using the HTC Vive tracker and log its position with much greater accuracy than the Black Room method. It is important to note that although this method allows us to evaluate the SLAM system outside of a simulation with greater accuracy, it is still performed in a controlled lab environment and does not account for many other sources of real-world noise which can affect the SLAM system, especially in outdoor environments. Another limitation of this setup is in the limited range of movement provided by HTC Vive’s “play area”. One way we are looking to expand this area of the testing is by using a Vicon Motion Capture system, that should provide more flexibility, more accuracy and bigger areas.

Outdoor

Following the HTC Vive test rig, we are investigating other methods of testing our SLAM system to account for more realistic scenarios of real-world implementations. This includes testing our system on a drone and setting up a camera rig on a car to collect outdoor data. However, obtaining the trajectory of the camera accurately in outdoor conditions can be difficult as GPS systems are typically accurate within a 5 meter radius, and this accuracy gets much worse near buildings and other obstacles. Obtaining accurate ground truth data in outdoor conditions is a widely open problem, and we are working on finding innovative solutions for it.

SLAM Accuracy in Academic Work

In the academic world, when measuring the performance of their SLAM system, the only parameter that is of concern is accuracy. For robustness on real world data, there is less concern as this can be optimised on a small set of sequence such as EuroC or KITTI. Furthermore, run time accuracy is of minimal concern, as long as it's approximately equivalent to real-time values. This is where the world of academia and industry differ massively.

A Note on Academic Sequences

After publication third parties have found issues in academic sequences commonly used for the evaluation of SLAM systems. Where possible we have corrected for these issues in our results for both KudanSLAM and other tested SLAM systems.

EuRoC[1]

The paper describing SOFT-SLAM[2] identified an issue in the EuRoC sequences:

> "We have additionally noted that the publicly available calibration parameters given for the V2 set of the tracks are wrong. A clear indicator for this was that the epipolar lines in these tracks were not aligned. Therefore, we had to apply an additional rotation of 0.24 deg between cameras to the original calibration files to align epipolar lines and achieve the results presented in the Table 1."

KITTI[3]

The paper describing IMLS-SLAM[4] identified an issue in the KITTI sequences:

> "First, we found a distortion of the scan point clouds because of a bad intrinsic calibration (we did a calibration of the intrinsic vertical angle of all laser beams of 0.22 degrees using the training data). Second, we found big errors in GPS data (used as ground truth) with, for example, more than 5 m in the beginning of sequence 8."

[1] The EuRoC micro aerial vehicle datasets - Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart

[2] SOFT-SLAM: Computationally Efficient Stereo Visual SLAM for Autonomous UAVs - Igor Cvisic, Josip Cesi, Ivan Markovic, and Ivan Petrovic

[3] Vision meets Robotics: The KITTI Dataset - Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun

[4] IMLS-SLAM: scan-to-model matching based on 3D data - Jean-Emmanuel Deschaud

Key Performance Indicators

At Kudan, to ensure that our continued development continues to improve and enhance our SLAM system, we monitor various aspects of the SLAM system, which we term Key Performance Indicators (KPIs). There are a number of KPIs which we measure:

- Accuracy
- Runtime
- Robustness
- Scalability
- Stability
- Complexity
- Portability
- Usability

New features can often come with a cost, for example it is almost always possible to sacrifice runtime for greater accuracy. Measuring these performance indicators is crucial for ensuring that during the design and implementation of new features the cost is not unwarranted. These measures ensure that the SLAM system we deliver to our customers does not only improve in terms of the features it supports, but also that performance is never compromised.

Accuracy

This can be measured in many ways, the Absolute Trajectory Error (ATE) measure is one of the most widespread ones and the one that we currently adopt. It assumes we have a ground-truth trajectory of the visual camera path, that we can extract the final SLAM trajectory, and that we can bring those to the same coordinate frame. If those assumptions are met, it measures the RMSE of the positional difference between SLAM trajectory positions and ground truth positions on a frame-by-frame basis. This is a good measure of the overall SLAM accuracy, because high accuracy in the trajectory is a necessary condition for other aspects of the SLAM output, like the 3D point map accuracy. If the trajectory is not accurate, the points will necessarily be not accurate as well.

To test accuracy we attempt to align the trajectory provided by KudanSLAM with the ground truth trajectory of the sequence. Ground truth trajectories are normally provided with academic test sequences. In the provided results we have used the well known EuRoC and KITTI test sequences to compare the accuracy of KudanSLAM and ORB-SLAM2 in both stereo and monocular modes. When evaluating accuracy results smaller is better, our results clearly show superior accuracy is provided by KudanSLAM.

Runtime

The runtime speed is the capability of the SLAM system to perform efficiently on a given hardware platform. Since SLAM is a real-time system, it needs at the very least to be able to process the input data faster than the sensor produce it. This is not enough, because SLAM also needs to spend time to maintain and optimise the map. Having low frame processing time allows for more computation time dedicated in cleaning and improving the map, with important repercussions on the quality of the output. Various measures can be employed, including overall CPU time usage, as reported from the OS, but the preferred one is to measure the time taken to call the *processFrame* function. This doesn't measure the overall computation, but combined with the FPS information about the input speed it allows understanding how much time the system spends relatively tracking and mapping.

To test runtime we time how long KudanSLAM takes to process sensor input each frame over the course of a sequence. The provided results highlight the significant speedup achieved by KudanSLAM over ORB-SLAM2. Another important measure is the maximum time taken to process a frame, again, shown in the results the maximum runtime of KudanSLAM is much lower and very stable over the course of a sequence, while ORB-SLAM2 can slow down by more than a factor of four on some sequences.

Robustness

This measures the capability of the SLAM system to keep tracking. SLAM is almost always non-deterministic, due to its multi-threaded process and OS' scheduler non-determinism. Given a sequence, multiple runs are performed during tests, and failures in tracking are recorded. The importance of this performance indicator is self-evident.

To test robustness we monitor the state of KudanSLAM's tracking performance for every frame of a sequence. The robustness score of a sequence is the ratio of the number of frames successfully tracked against the total number of frames in a sequence. The provided results show similar results between KudanSLAM and ORB-SLAM2 on simpler sequences but as the sequences get more difficult KudanSLAM's performance is superior.

Scalability

This refers to the memory usage in time, and to the ability to use SLAM on long sequences and big maps. By definition, SLAM is unbounded in memory usage, because it must keep track of all the map, the size of which is dependent on the input. However, different SLAM system use the memory more or less efficiently given the same input, and this performance indicator measures just that against a set of given input sequences.

To test scalability we measure the memory usage of KudanSLAM over the course of a sequence. This value is measured by the operating system rather than an internal tool to ensure it is as close as possible to the full amount of memory required. The provided results show that KudanSLAM uses substantially less memory than ORB-SLAM2 over the same sequences. Additionally, as KudanSLAM reuses the memory of previously mapped areas, the memory usage is only tied to the size of the map not the duration of the sequence.

Stability

The stability describes how consistently KudanSLAM performs given the same input. Most modern SLAM systems are non-deterministic and will produce different results for the same sensor input based on how the limited processing power was distributed between the decoupled frontend and backend of the system. Currently KudanSLAM falls into this category and its stability describes the variance of the results for the same sequence.

Due to the non-deterministic nature of the slam systems we provide box and whisker plots that show the maximum, minimum, quartiles and average results over multiple runs of a sequence. The provided results show that the worst case and deviation of KudanSLAM is significantly smaller than ORB-SLAM2 showing our much greater stability.

Complexity

This measures how the computational time for map processing increases as the map size grows. Care must be taken in keeping the map processing efficient as more keyframes interconnect with each other.

Portability

Simply measures how many different SLAM features are available in how many different hardware platforms, operating systems and sensor configurations. The tables in previous sections shows how we measure it.

Usability

This is a more qualitative measure on the complexity of the APIs and measures both the ratio of functions over features, and the number of support messages relative to APIs we get with respect to active users of the codebase.

Results

At Kudan we are devoted to accurate and rigorous testing. We will soon be releasing more comprehensive test results for the below scenarios:

● More SLAM Systems:	Such as DSO and VINS-Fusion.
● More Input types:	Such as Mono, RGBD, and IMU.
● More Sequences:	Such as TUM, and custom difficult sequences.
● More Hardware:	Such as Macbooks, iPads, and Raspberry Pis.

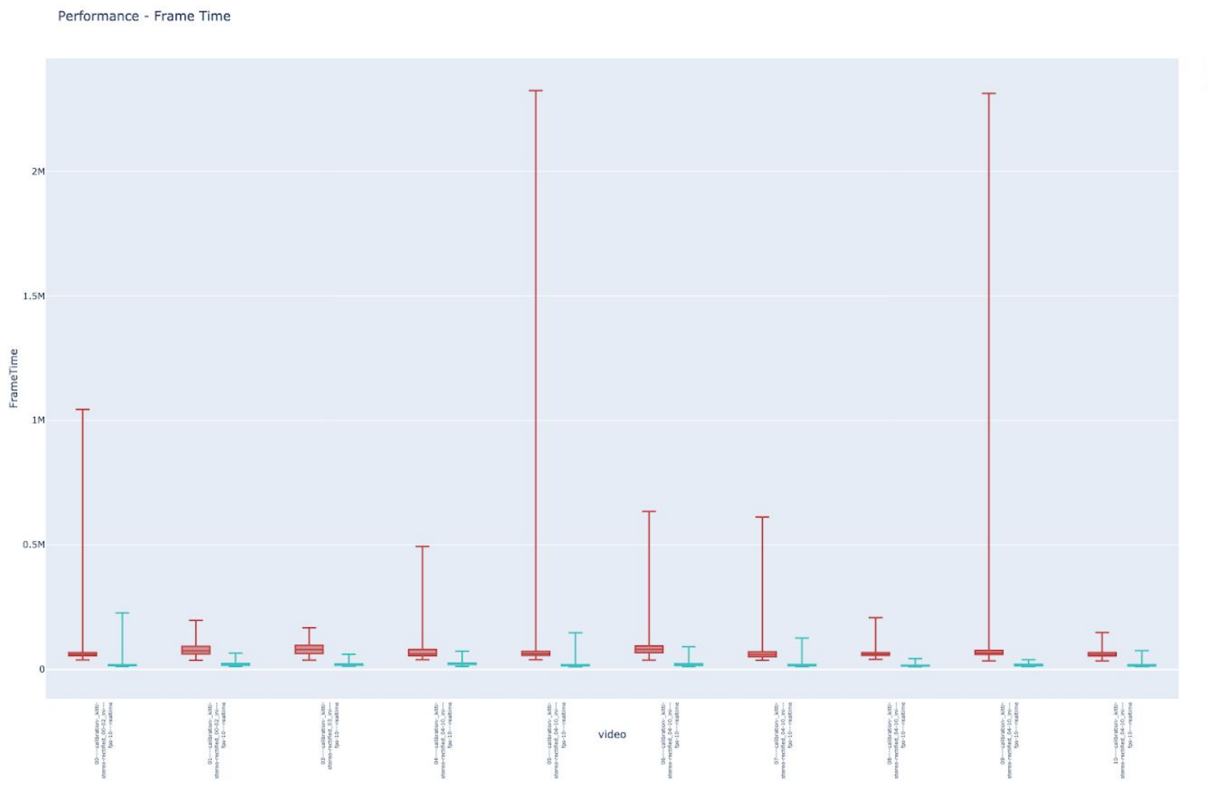
Until then we have included a number of figures comparing the performance of KudanSLAM with ORB-SLAM2 on the EuRoC and KITTI datasets. These tests were run on new c5.12xl, c5.24xl, and c5.metal instances feature custom 2nd generation Intel Xeon Scalable Processors (Cascade Lake) using two cores with a sustained all core Turbo frequency of 3.6GHz and single core turbo frequency of up to 3.9GHz.

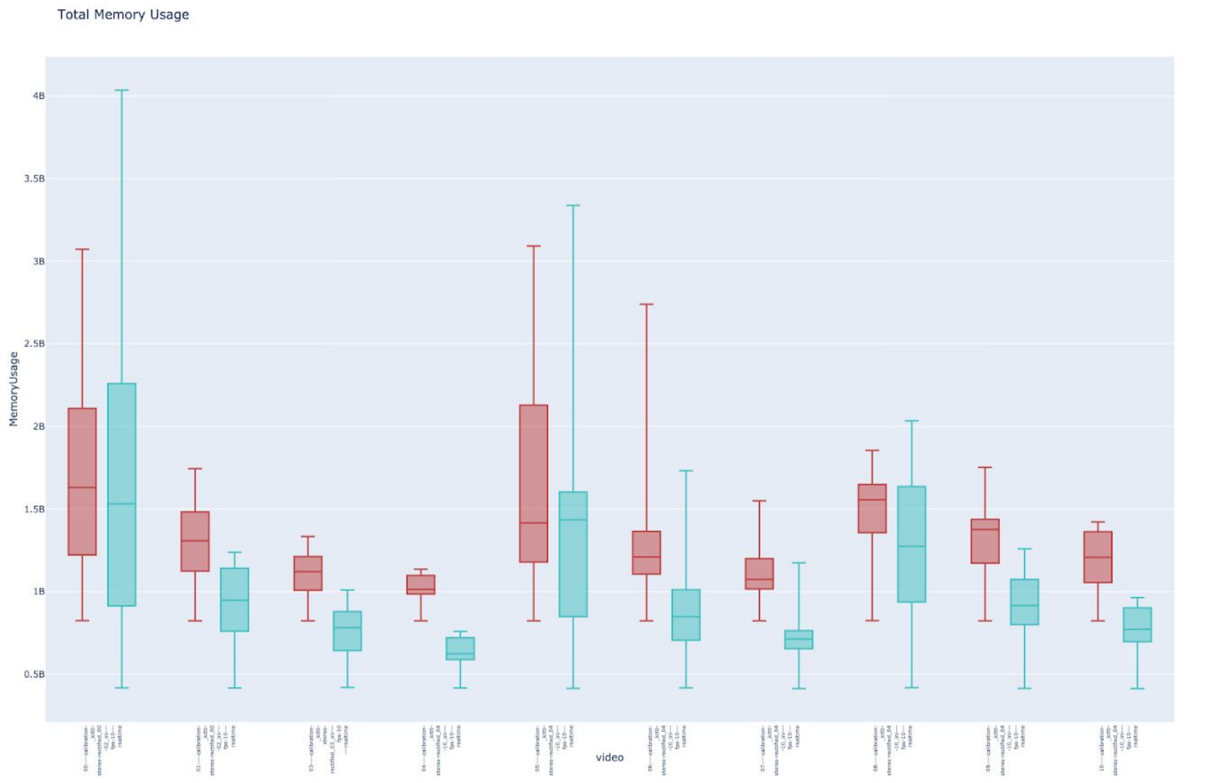
Current list of results:

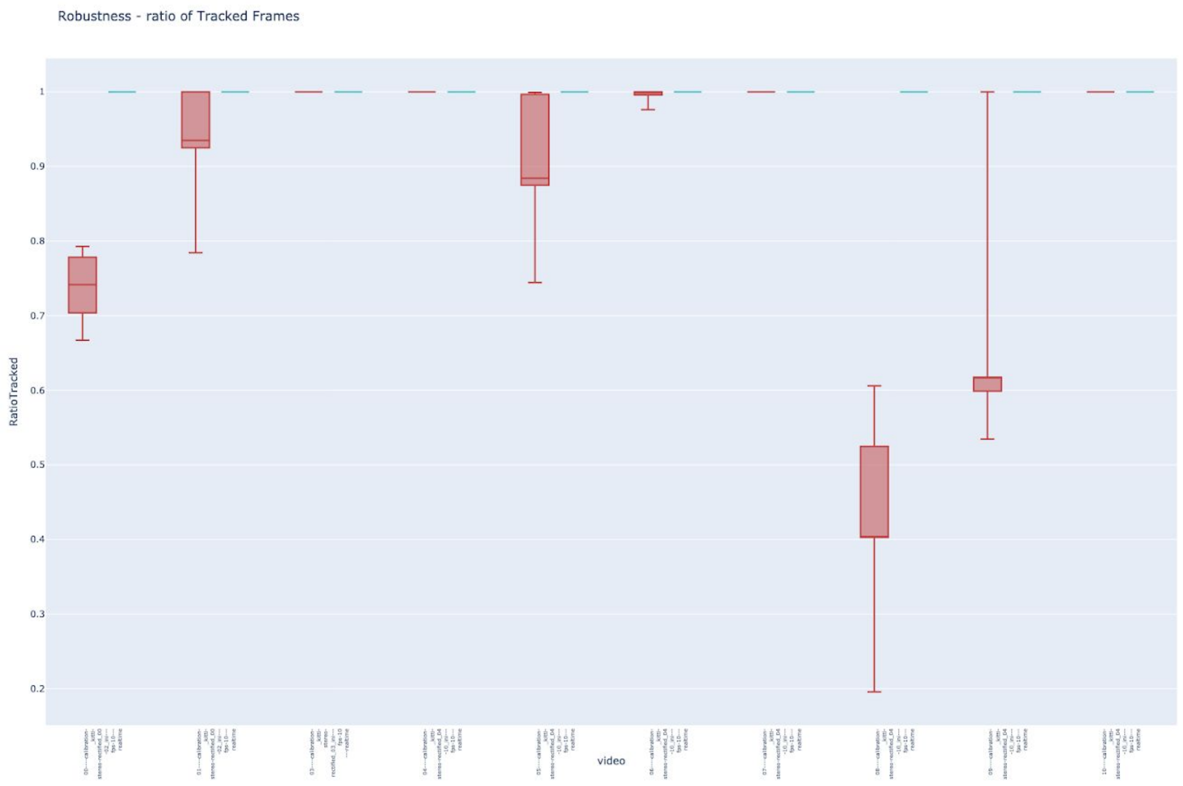
- KITTI Stereo KudanSLAM vs Stereo ORB-SLAM2
- EuRoC Stereo KudanSLAM vs Stereo ORB-SLAM2
- EuRoC Mono KudanSLAM vs Mono ORB-SLAM2
- EuRoC Mono KudanSLAM vs Stereo KudanSLAM

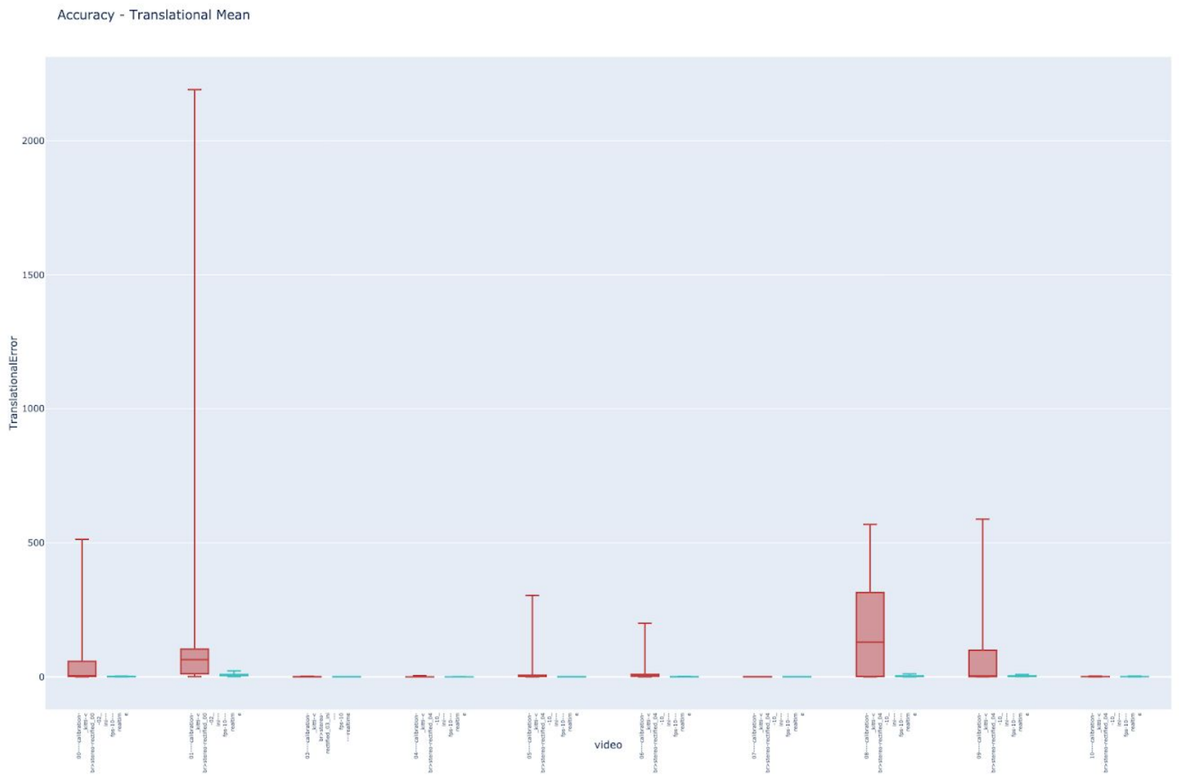
KITTI Stereo KudanSLAM vs Stereo ORB-SLAM2

Red - ORB-SLAM2   Blue - KudanSLAM

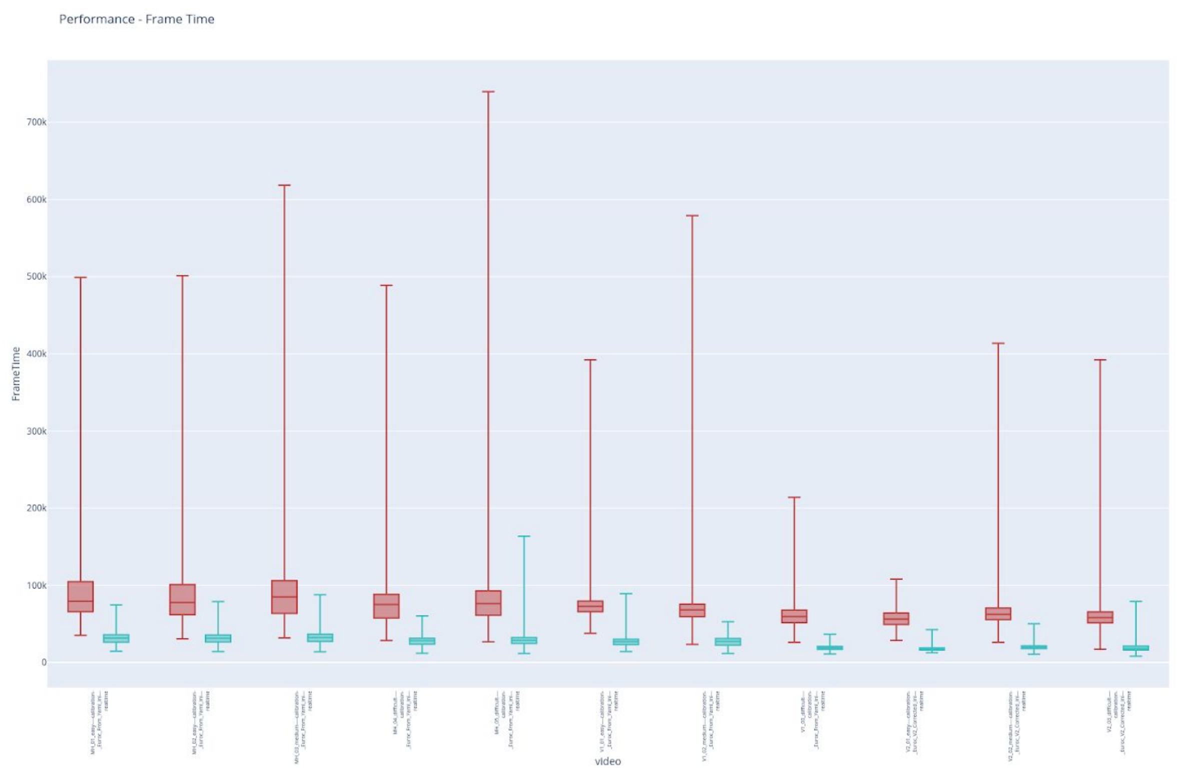


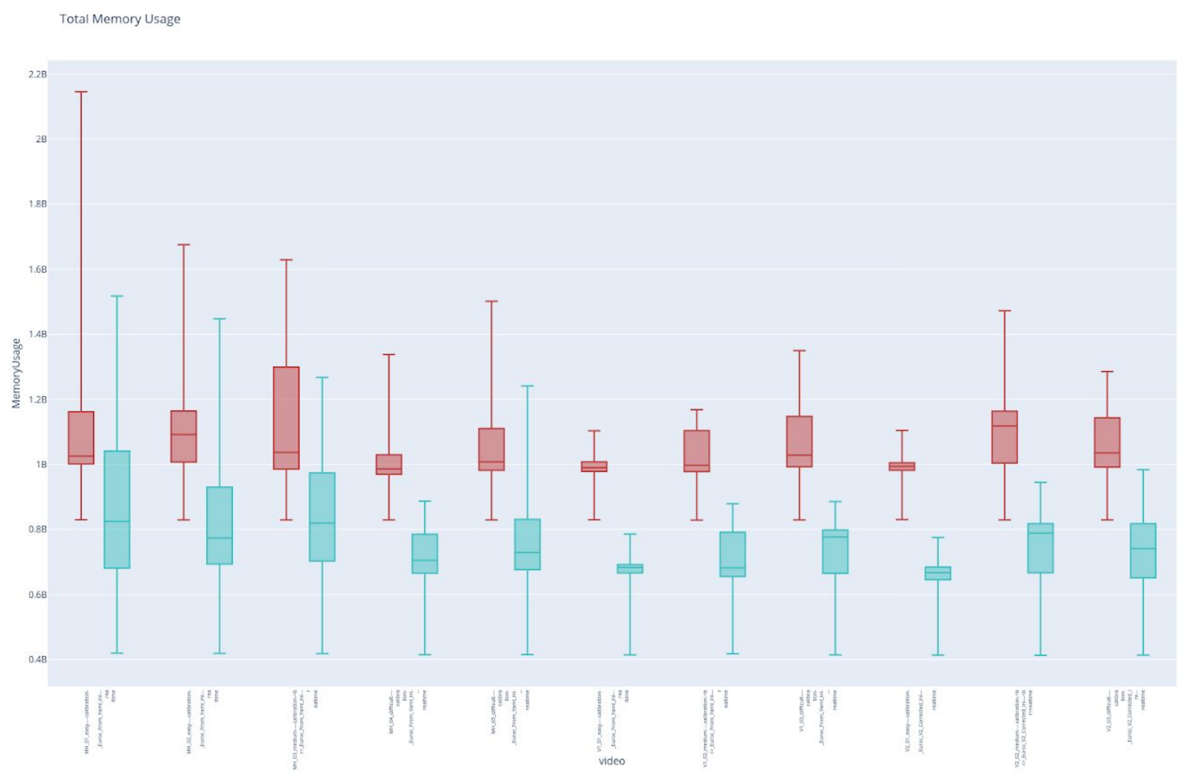


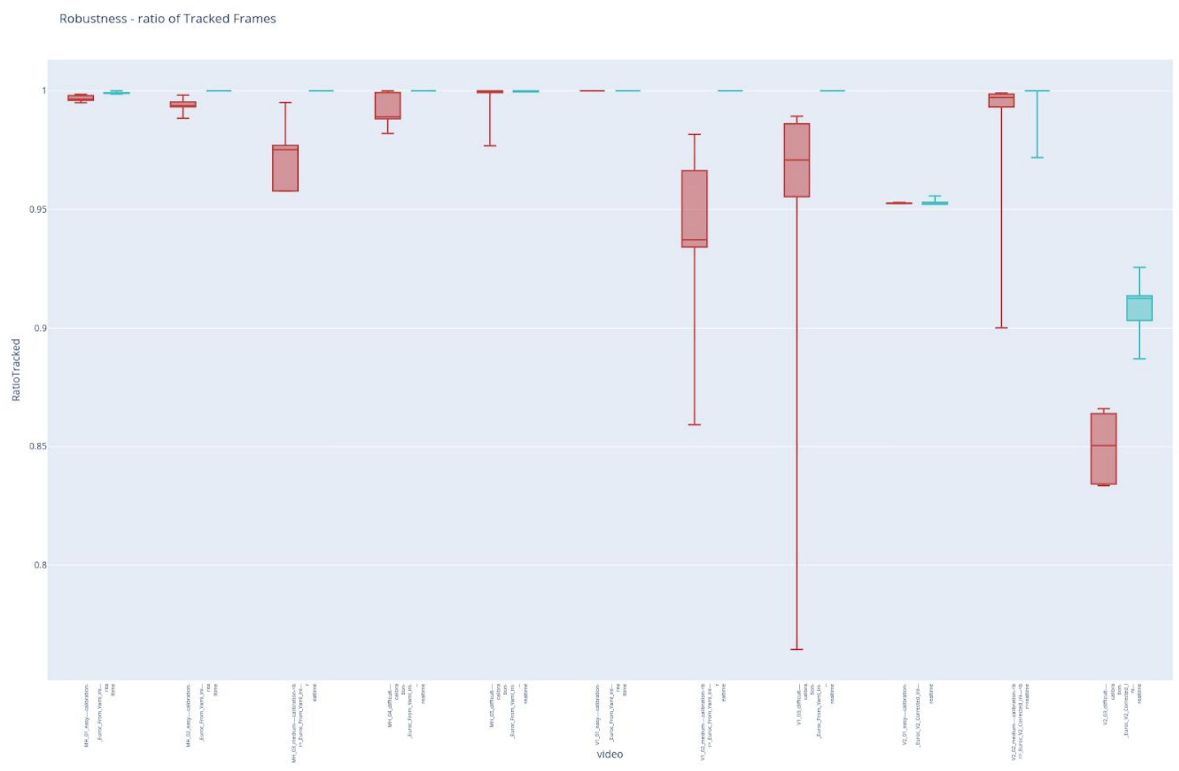

Robustness - ratio of Tracked Frames
RatioTracked
video
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2

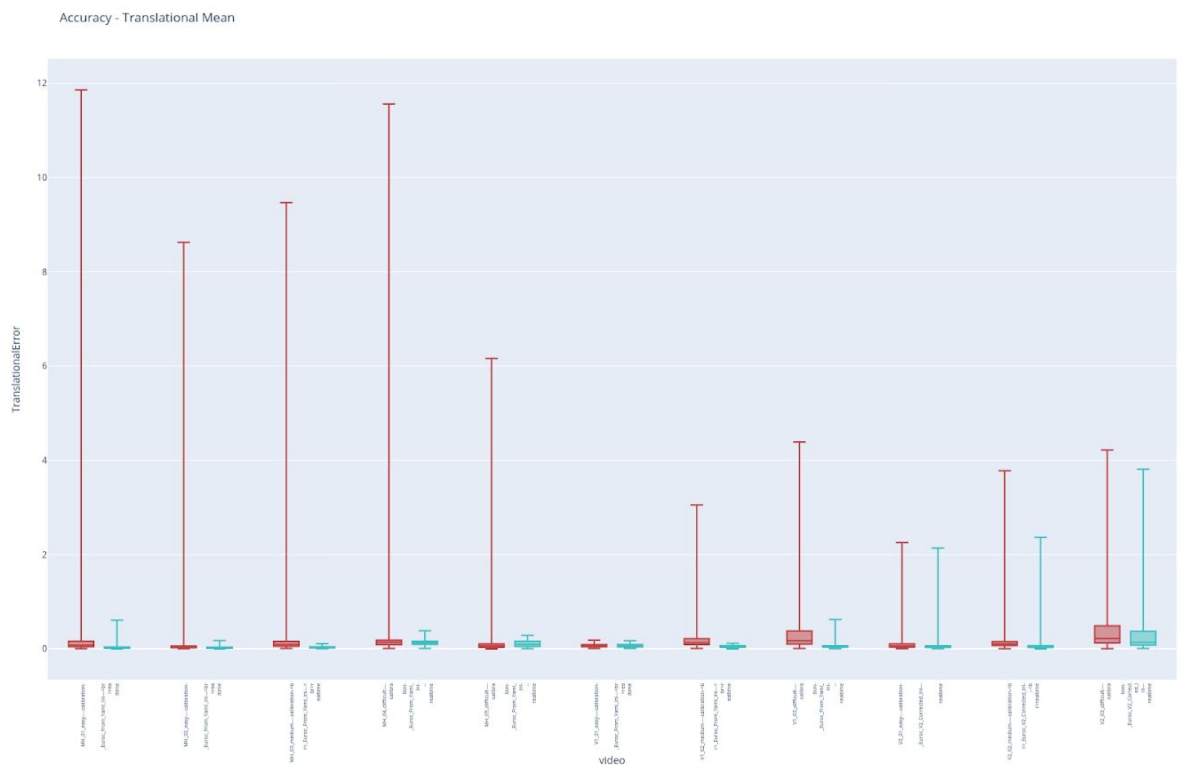

Accuracy - Translational Mean
TranslationalError
video
2000
1500
1000
500
0

EuRoC Stereo KudanSLAM vs Stereo ORB-SLAM2



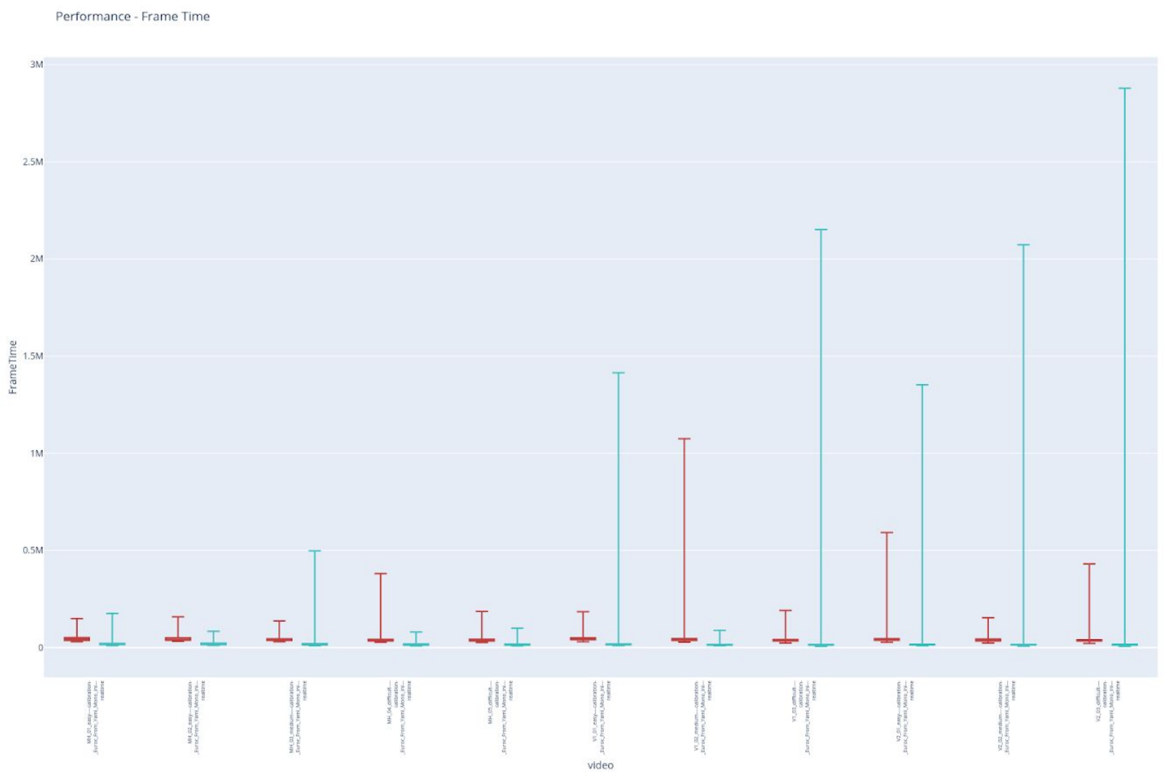


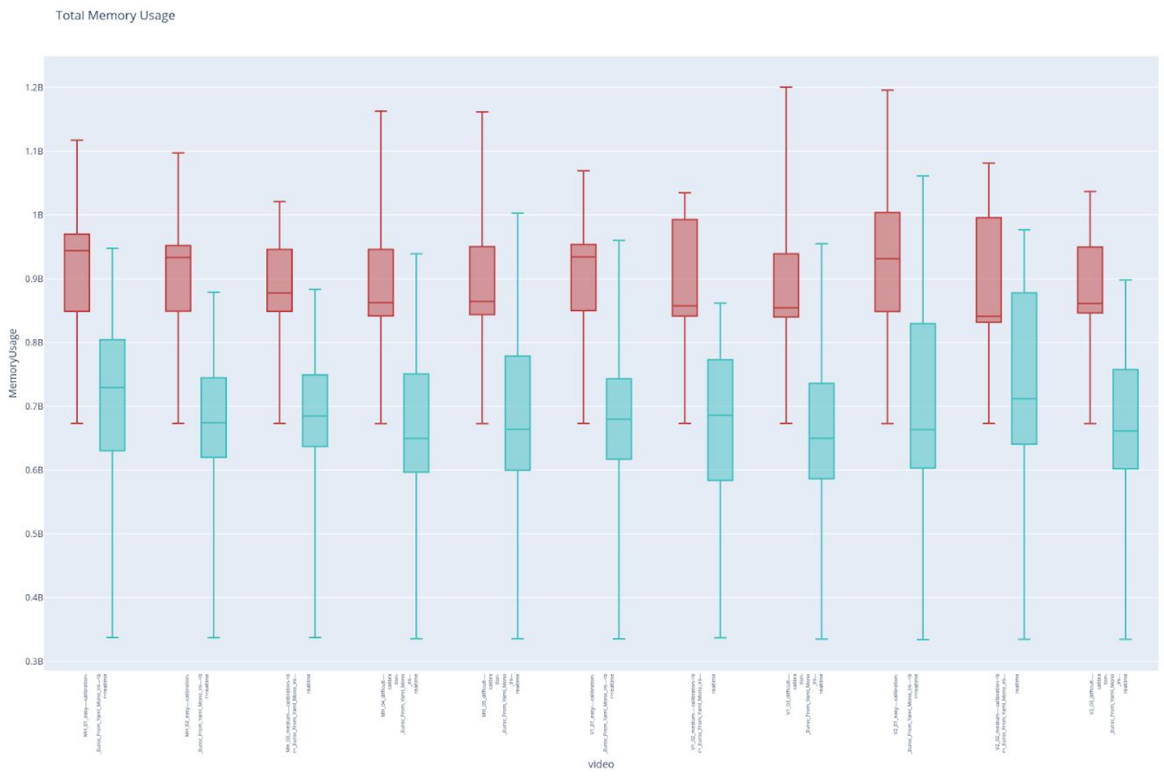

Robustness - ratio of Tracked Frames
RatioTracked
1
0.95
0.9
0.85
0.8
video
MH_01_easy---calibration-_Euroc_From_Yaml_ini---realtime
MH_02_easy---calibration-_Euroc_From_Yaml_ini---realtime
MH_03_medium---calibration-cam_Euroc_From_Yaml_ini---realtime
MH_04_difficult---calibration-_Euroc_From_Yaml_ini---realtime
MH_05_difficult---calibration-_Euroc_From_Yaml_ini---realtime
V1_01_easy---calibration-_Euroc_From_Yaml_ini---realtime
V1_02_medium---calibration-cam_Euroc_From_Yaml_ini---realtime
V1_03_difficult---calibration-_Euroc_From_Yaml_ini---realtime
V2_01_easy---calibration-_Euroc_V2_Corrected_ini---realtime
V2_02_medium---calibration-cam_Euroc_V2_Corrected_ini---realtime
V2_03_difficult---calibration-_Euroc_V2_Corrected_ini---realtime

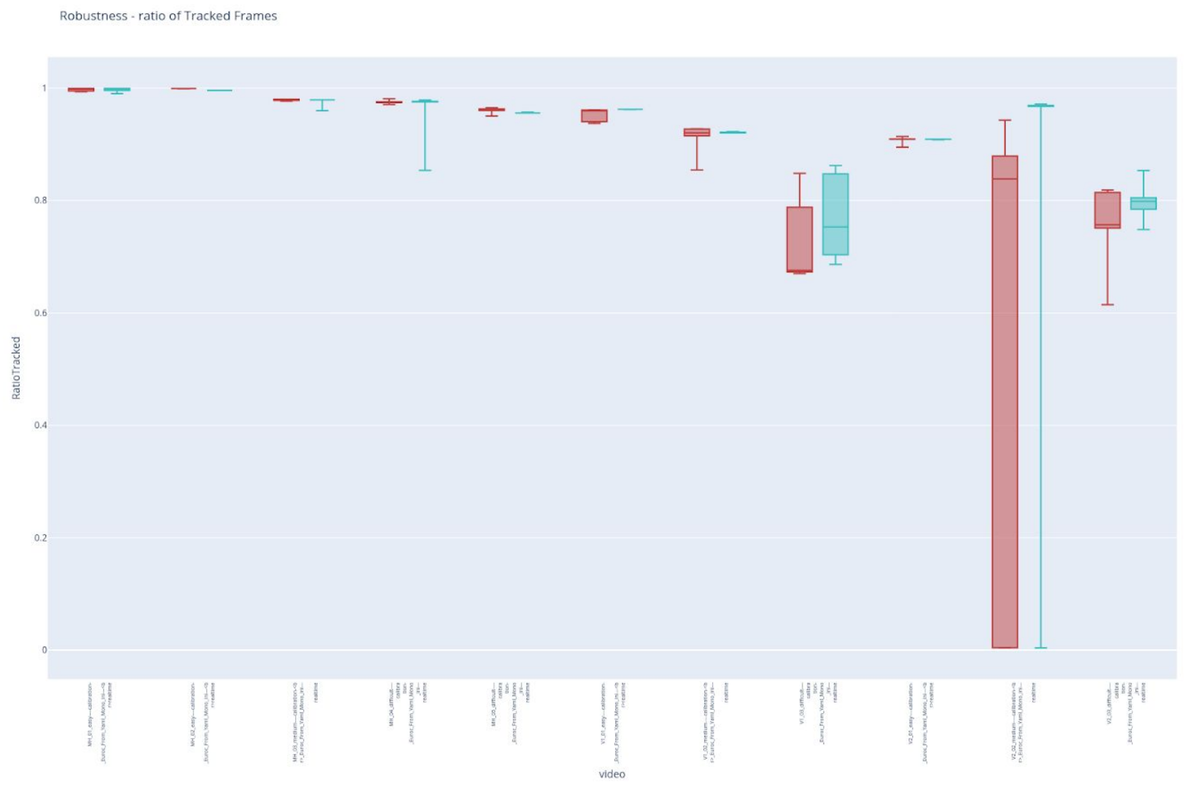

Accuracy - Translational Mean
TranslationalError
12
10
8
6
4
2
0
video
MH_01_easy---calibration-_Euroc_From_Yaml_ini---realtime
MH_02_easy---calibration-_Euroc_From_Yaml_ini---realtime
MH_03_medium---calibration-cam_Euroc_From_Yaml_ini---realtime
MH_04_difficult---calibration-_Euroc_From_Yaml_ini---realtime
MH_05_difficult---calibration-_Euroc_From_Yaml_ini---realtime
V1_01_easy---calibration-_Euroc_From_Yaml_ini---realtime
V1_02_medium---calibration-cam_Euroc_From_Yaml_ini---realtime
V1_03_difficult---calibration-_Euroc_From_Yaml_ini---realtime
V2_01_easy---calibration-_Euroc_V2_Corrected_ini---realtime
V2_02_medium---calibration-cam_Euroc_V2_Corrected_ini---realtime
V2_03_difficult---calibration-_Euroc_V2_Corrected_ini---realtime

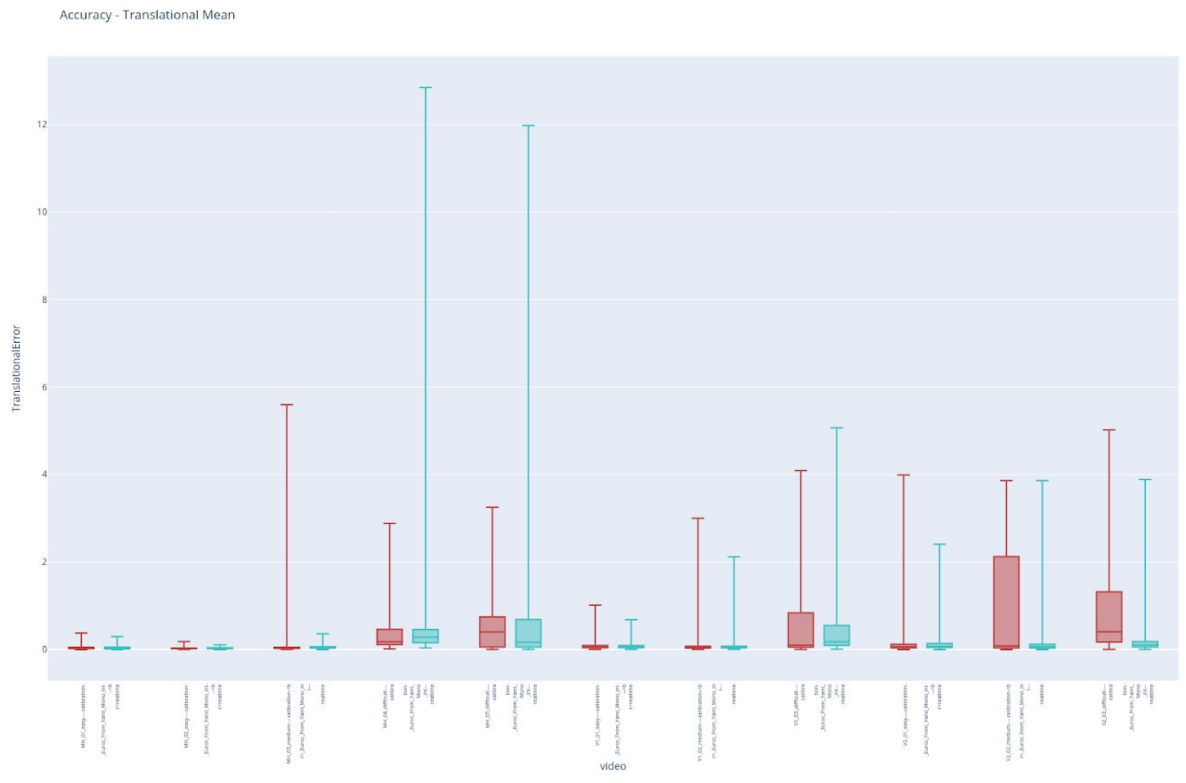
EuRoC Mono KudanSLAM vs Mono ORB-SLAM2

Red - ORB-SLAM2   Blue - KudanSLAM



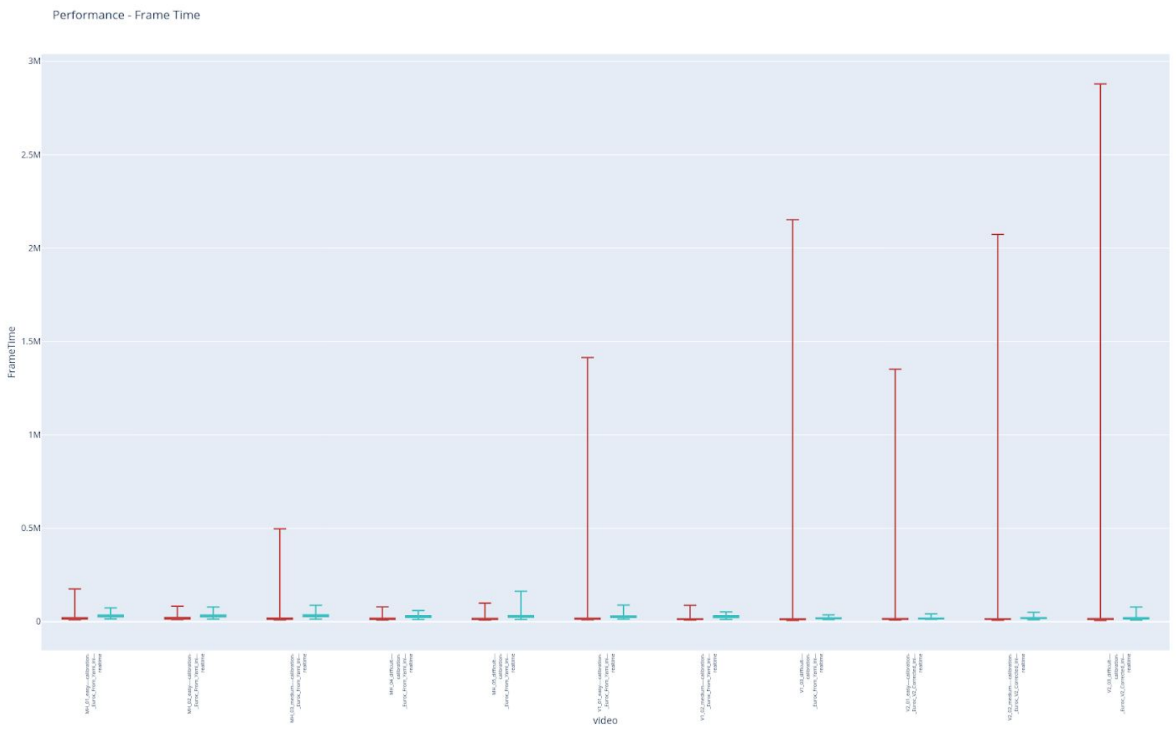


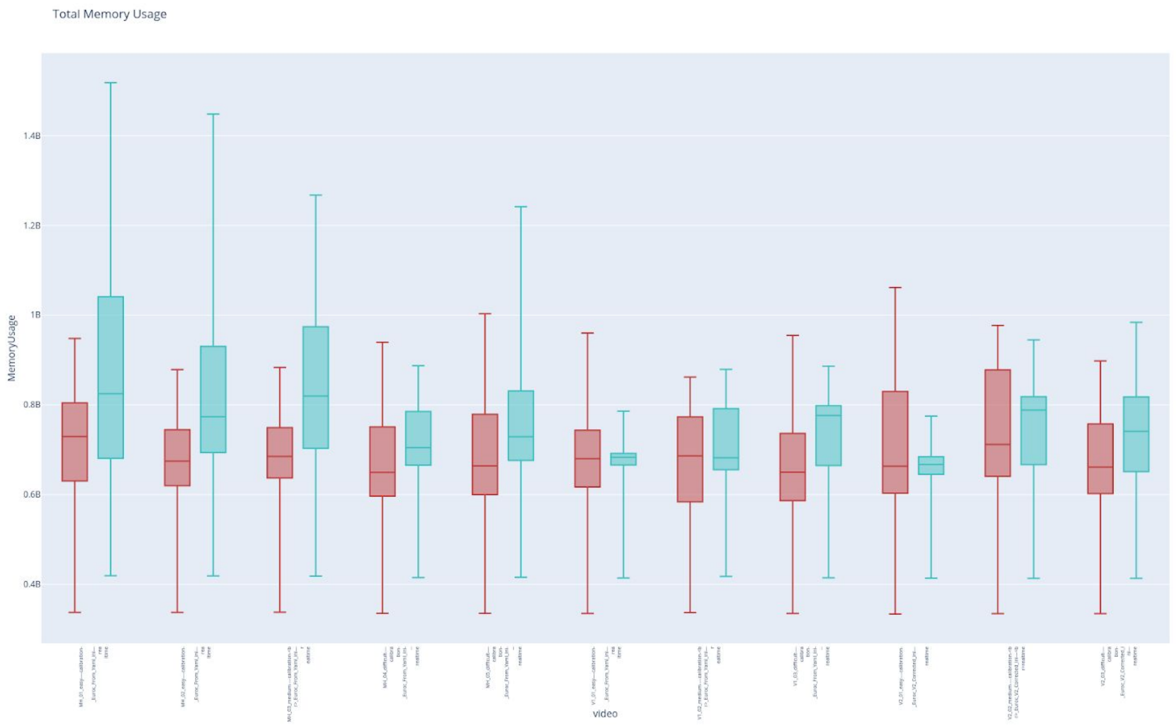

Robustness - ratio of Tracked Frames
RatioTracked
1
0.8
0.6
0.4
0.2
0
video


Accuracy - Translational Mean
TranslationalError
12
10
8
6
4
2
0
video

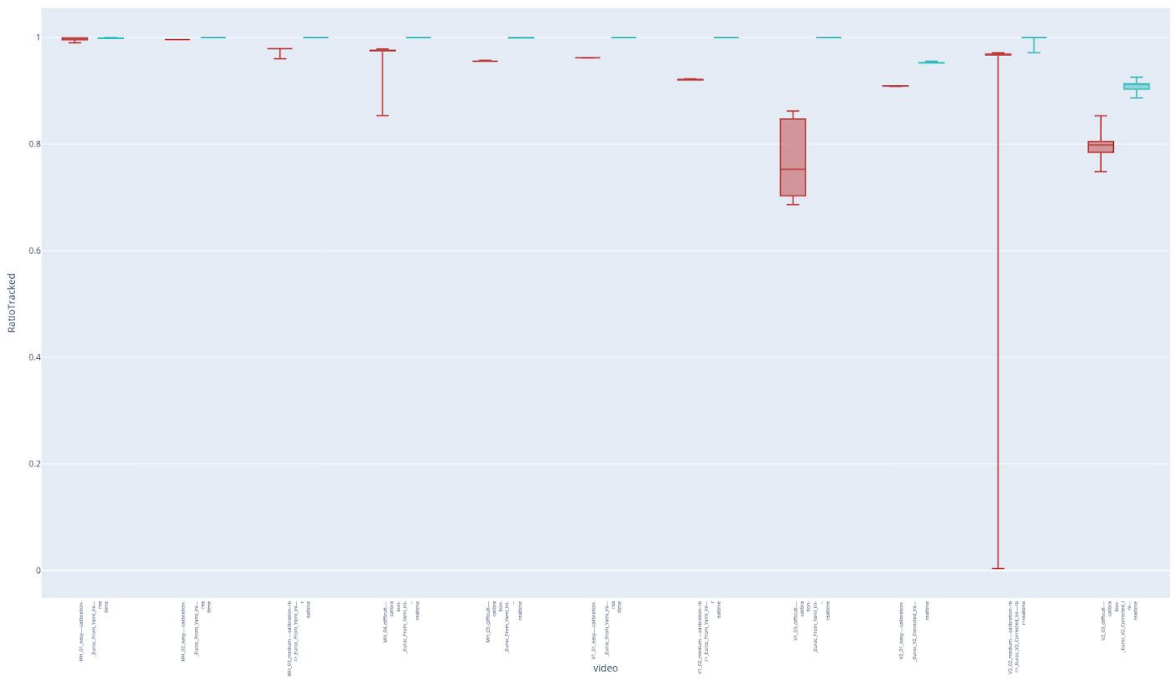
EuRoC Mono KudanSLAM vs Stereo KudanSLAM

Red - Mono   Blue - Stereo

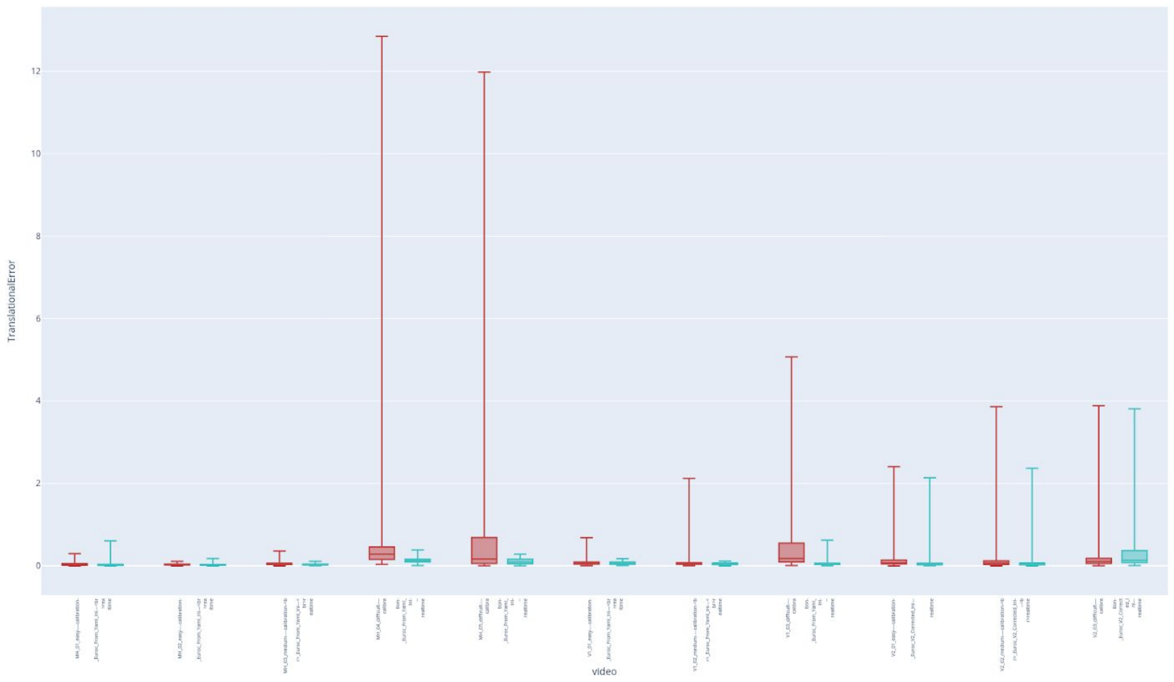




Robustness - ratio of Tracked Frames



Accuracy - Translational Mean



Relocalisation

To test relocalisation performance we first map an area but running a sequence through KudanSLAM. Once we have created a map we disable the mapping system, so the map will not be expanded, and force KudanSLAM into a lost state. By changing the sequence we can test relocalisation performance against the previously generated map.

The results tables 5 and 6 show the relocalisation performance of every frame of a number of sequences taken in the same environment. In these experiments the initial sequence used to generate the map was MH_01, therefore relocalisation performance on this test sequence should be perfect. The results from the ORB-SLAM2 approach (Table 5) clearly show issues in their approach, which KudanSLAM (Table 6) does not suffer from. The reduced relocalisation performance of KudanSLAM on sequence MH_03 is caused by the different trajectory taken by this sequence compared to MH_01. The different trajectories resulted in no overlap between the map and the test sequence for large numbers of frames.

Video	Total Frames	Relocalised Frames	Success
MH_01	738	431	58.4%
MH_02	612	188	30.7%
MH_03	544	183	33.6%

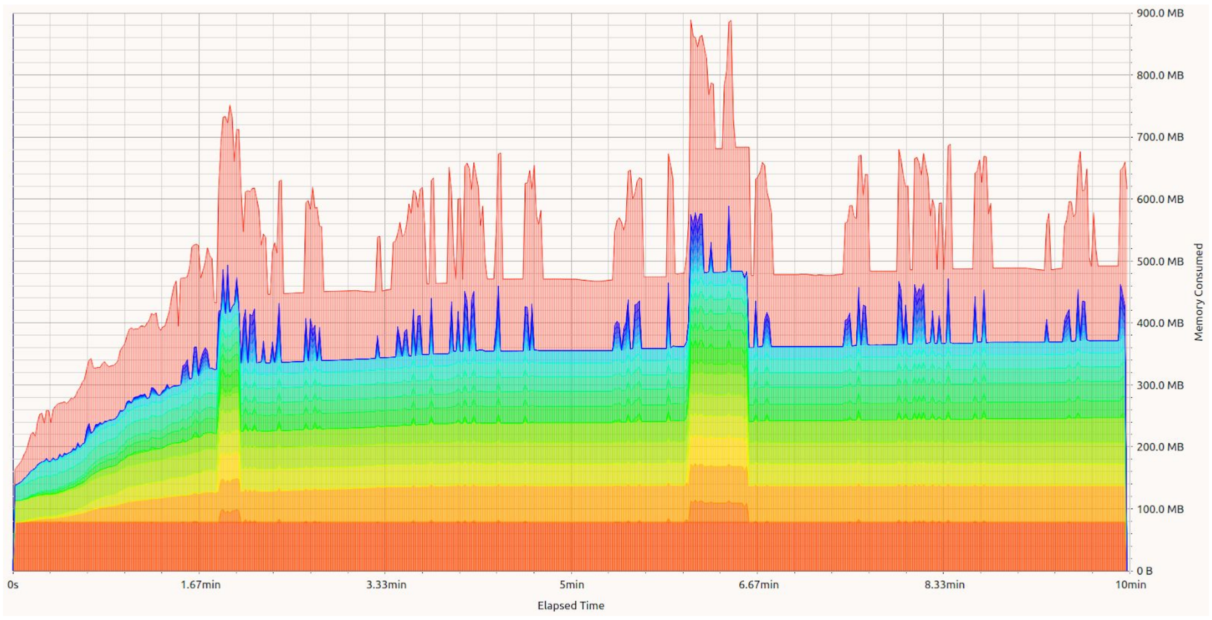
Table 5 - Relocalisation results based on ORB-SLAM2

Video	Total Frames	Relocalised Frames	Success
MH_01	738	738	100.0%
MH_02	612	586	95.8%
MH_03	544	303	55.7%

Table 6 - Relocalisation results using KudanSLAM

Memory Over Time

KudanSLAM reuses the map when revisiting the same areas to limit both drift and memory consumption. To validate the map reuse and ensure that memory consumption is tied to the map size and not the duration of a sequence we repeatedly run the same sequence a number of times. The results below are from running the MH_05 sequence five times in succession. As shown by the figure memory usage increases as the map grows during the first loop and then settles at approximately two minutes when the second loop begins.



Inertial Measurement Units (IMUs)

We are currently working on producing results graphs for both the monocular plus IMU and stereo plus IMU configurations. The addition of an IMU to a visual slam system enables some features which are not possible without one, first in monocular an IMU enables the system to track the scale of the map, and second in both monocular and stereo an IMU enables orienting the map with respect to gravity. As IMUs are not dependent on the scene around the camera(s) they can also increase the robustness of the system. The situation where the camera(s) do not detect any visual features is quite possible in some environments and by utilising an IMU the system can maintain tracking even without any visual information for a short time.

Stereo vs Mono Cameras

The results show that SLAM with a stereo camera is more robust than with a monocular camera in all sequences. Similarly, stereo camera result performs better in terms of accuracy and stability. This is partly due to monocular SLAM performing poorly when mapping pure rotation as parallax is required to triangulate points, which is provided by the second camera in a stereo system.

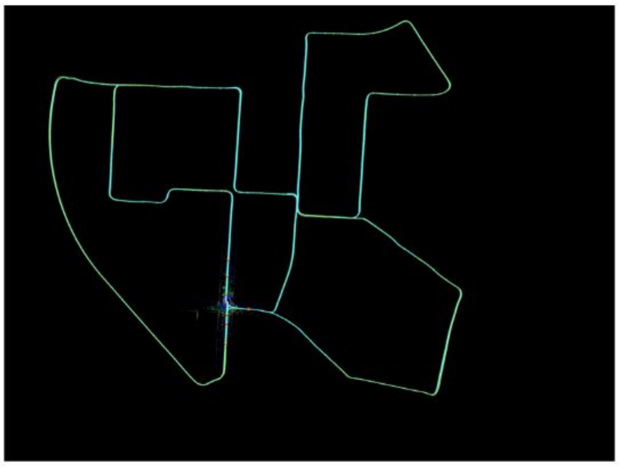
Another disadvantage of monocular slam is scale drift, it is not possible to recover metric scale from a monocular sequence without additional information, for example GPS or IMU. Stereo cameras can use the parallax between the cameras to compute scale at all times and therefore produce metrically scaled maps and do not suffer from scale drift.

Finally, one downside of stereo SLAM is the need to process two images, this generally gives monocular SLAM the advantage in per frame process time. As monocular SLAM only processes one image the map also stores less information per frame and therefore monocular slam normally uses less memory and is therefore more scalable.

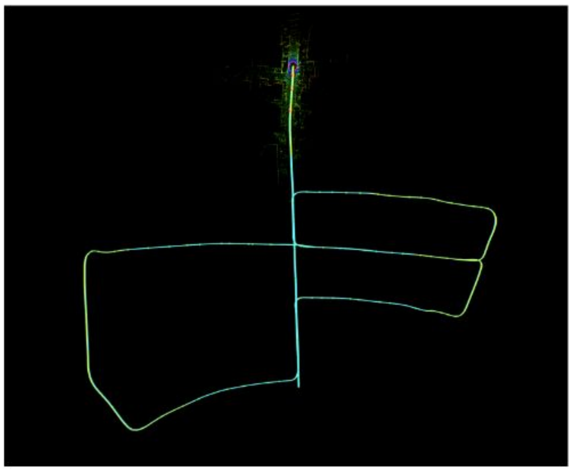
LiDAR Mapping

As we advance KudanSLAM and add the ability to fuse the information provided by more sensors our system becomes both more sensor agnostic and more reliable. The below image is an early peek at the LiDAR mapping integration. Coming soon.

KITTI sequences, Cyan = Kudan, Yellow = ground truth



Sequence 0



Sequence 5

KudanSLAM Visual and LiDAR fusion has a localisation accuracy of approximately 0.01% of the driving distance on the KITTI data set. In metric units this is a drift of about 0.1m over a 1000m drive.