AARHUS UNIVERSITET

Institut for Elektro- og Computerteknologi

Bachelor Project
Comparison of Monocular Visual-Inertial Odometry in Aerial Drones

Mads Evander Jensen MEJ Student no. 202107547 AU-id: AU705673
Jeppe Emil Smedegaard Pape JESP Student no. 202008596 AU-id: AU682984

Supervisor: Andriy Sarabakha

DATE
May 22, 2025

Number of characters: XXX with spaces

1 Abstract

Monocular Visual-Inertial Odometry is the process of estemating the position and velocity state of a robot in respect to the surroundings, using sensors(inertial) and a single camera(monocular). In this project we are only interested in odometry in aerial drones, but some of the findings are also relevant for other drones. In this project we implement and test multiple methods of odometry on an aerial drone. Our findings show that ... is curently the best method for estemating an aerial drone' state.

2 Preface

A special thanks to our supervisor Andriy Sarabakha for overseeing our research and borrowing us the necessary materials and to the AU Department of Electrical and Computer Engineering for letting us borrow an office space and grant access to reseach facilities.

Abbreviation	Full Form
AU	Aarhus University
LAB	Laboratory
DEV	Developer
VR/VR	Virtual- /Augmented Reality
CPU	Central Processing Unit
RAM	Random-Access Memory
ADB	Android Debug Bridge
MAV	Micro Air Vehicle
VIO	Visual-Inertial Odometry
SLAM	Simultaneous Localization And Mapping
IMU	Inertial Measurement Unit
ROS	Robot Operating System
FAST	Features from Accelerated Segment Test
BRIEF	Binary Robust Independent Elementary Features
ORB	Oriented FAST and rotated BRIEF
ATE	Absolute Trajectory Error

Table 1: List of abbreviations used in this project

Contents

1 Abstract

2 Preface

3 Introduction **1**
3.1 Motivation . . . 1
3.2 Procedure . . . 1
3.3 Research Objective . . . 1
3.4 Thesis Structure . . . 1

4 Background **2**
4.1 Visual-Inertial Odometry . . . 2
4.2 Benchmarking Datasets . . . 3
4.2.1 EuRoC dataset . . . 3
4.2.2 AUDrone dataset . . . 3

5 Methodology **5**
5.1 Research Approach . . . 5
5.1.1 Setup . . . 5
5.1.2 Evaluation metrics . . . 5
5.2 Selection of VIO methods . . . 5

6 System Setup and Configuration **7**
6.1 Configuration of VIO systems . . . 7
6.2 Data Preparation and Preprocessing . . . 9
6.3 AU Drone Setup and Calibration . . . 9

7 Testing **12**
7.1 AU drone . . . 12
7.2 Evaluation Metrics . . . 12
7.3 Testing Procedure and Reproducibility . . . 12
7.3.1 Automation of programs on all 11 bags . . . 12
7.3.2 Automation of trajectory data conversion . . . 13
7.3.3 Automation of Evo evaluation with the groundtruths . . . 13

8 Results **15**
8.1 Analytical comparison . . . 15
8.2 Table: Accuracy . . . 15
8.3 Table: CPU/Memory Usage . . . 17
8.4 Graphic comparison . . . 18

9 Discussion **19**
9.1 Key Findings . . . 19
9.2 Strengths and Weaknesses of Evaluated VIOs . . . 19
9.3 Challenges Encountered During Research . . . 19
9.4 Limitations of the Study . . . 20
9.5 Lessons Learned . . . 20

10 Conclusion **21**

11 Attachments **23**
11.1 PX4 developer kit - kalibr calibration results . . . 23
11.2 Project Proposal . . . 25

3 Introduction

3.1 Motivation

The increasing demand for autonomous aerial drones across diverse applications like inspection, VR/AR, surveillance, and delivery, hinges on their ability to navigate reliably and accurately in complex environments. Monocular Visual-Inertial Odometry is one solution for enabling such autonomy by fusing data from a single camera and inertial measurement unit to estimate the drone's pose. However, the effectiveness of different VIO methods can vary significantly depending on factors such as the environment, available computational resources, and of course the specific algorithm implemented. In this thesis we will evaluate and compare the performance of various VIO methods. By analyzing their pose accuracy, computational, and memory usage across different datasets, this research aims to provide valuable insights into the practical deployability and suitability of these methods for achieving robust and efficient autonomous navigation in real-world aerial drone operations.

3.2 Procedure

To evaluate the performance of several state-of-the-art VIO methods, we will implement and test them on two distinct datasets: one captured in an AU laboratory setting to provide a controlled environment, and a publicly available dataset for broader comparison. Accuracy will be assessed by comparing the estimated pose data from each VIO method against corresponding ground truth measurements. Additionally, we will measure the computational and memory resources consumed during these tests to understand their practical efficiency and thereby their deployability.

- Identify, implement, and configure several state-of-the-art monocular Visual Odometry (VO) and Visual-Inertial Odometry (VIO) methods relevant to aerial robotics.
- Develop a comprehensive evaluation framework capable of assessing the performance of these methods across multiple metrics.
- Evaluate the accuracy and resource usage of the selected VO and VIO methods.

3.3 Research Objective

Our primary objective of this research is to evaluate and compare the performance characteristics of various monocular visual-inertial odometry methods. Focusing on key metrics essential for real-world application, this evaluation will determine their practical suitability and deployability for autonomous aerial drone navigation, ultimately serving as a guide for selecting methods that ensure robust and efficient autonomous flight.

3.4 Thesis Structure

The thesis is structured to guide the reader through our research process on comparing Visual-Inertial Odometry. We begin by providing the necessary Background, introducing VIO concepts and the benchmarking datasets used. This is followed by the Methodology, detailing the research approach. The practical aspects are covered in System Setup and Configuration, explaining how the VIO systems were configured, data prepared, and the testing framework implemented. The Testing section outlines the specific procedures, evaluation metrics, and methods employed to ensure reproducibility. The findings are then presented in the Results section. At last in the discussion we interpret the key findings and discusses the strengths and weaknesses of the evaluated systems, and reflects on challenges, limitations, and lessons learned. The thesis concludes with summarizing the work.

4 Background

4.1 Visual-Inertial Odometry

Monocular visual-inertial odometry (VIO) leverages data from a single camera and an inertial measurement unit (IMU) to estimate the position and orientation (pose) of a robot. While camera-based state estimation is accurate during low-speed navigation, it faces significant challenges such as motion blur, track loss, and scale ambiguity at higher speeds. In contrast, inertial navigation systems excel in high-speed scenarios and provide pose estimates at a world scale, but they may suffer from noise and drift over time. By combining the strengths of both sensors, visual-inertial odometry can achieve improved accuracy and robustness, making it suitable for a wide range of applications, including aerial robotics. Our research will specifically focus on monocular VIO. This limitation to single-camera platforms presents a harder problem compared to multi-camera setups, where algorithms can leverage the additional data from the second camera, and specifically the overlapping fields of view for improved accuracy, but this comes at increased computational usage and complexity.

The VIO methods we will test have different key aspects and features. One common feature is doing SLAM (Simultaneous Localization and Mapping), which is the general technique where a robot/camera builds a map of an unknown environment while tracking its own position within it. In this project we are only interested in the VIO part of SLAM. Though it should be considered that the VIO methods doing SLAM have extra computational overhead.

Another common feature is loop closure, which is the process of recognizing a previously visited location to correct accumulated drift. Over time, small errors in pose estimation accumulate, causing the drones pose to become inaccurate. It detects when the robot returns to a known place and adjusts the trajectory accordingly. We will include VIO methods that perform loop closing and some that don't. It will be interesting to see if there will be clear difference in the results.

To comprehensively evaluate VIO methods, we selected VIO methods with various feature detection and description algorithms, a crucial component of VIO. The most common feature detection algorithms include FAST, ORB and Shi-Tomasi methods.

- FAST is a corner detection algorithm known for its computational efficiency. It identifies potential corner points by examining a circular region of 16 pixels around a candidate pixel. A pixel is classified as a corner if a sufficient number of consecutive pixels on the circle are either significantly brighter or significantly darker than the candidate pixel itself. This method is much faster than many other corner detectors, making it suitable for real-time applications.
- ORB is another highly efficient and robust feature detection and description algorithm that builds on FAST. It first uses the FAST algorithm to detect keypoints in an image. Then, it applies a modified Harris corner measure to select the strongest keypoints. For each keypoint, ORB calculates its orientation and then computes a binary descriptor called rBRIEF. rBRIEF is an improvement over the original BRIEF descriptor, making it more robust to rotation by "steering" the sampling pattern according to the keypoint's orientation.
- Shi-Tomasi is an improvement upon the Harris corner detector. Both algorithms use the concept of an "autocorrelation matrix" to analyze how much the image intensity changes when a small window is shifted in different directions. While Harris uses a specific scoring function based on the eigenvalues (λ_1, λ_2) of this matrix ($R = \lambda_1\lambda_2 - k(\lambda_1 + \lambda_2)^2$), Shi-Tomasi simplifies this by defining a corner as a point where the minimum of the two eigenvalues ($\min(\lambda_1, \lambda_2)$) is above a certain threshold. [1]

Some VIO methods, like VI-DSO, opt for custom direct feature detection, meaning they directly utilize the raw pixel intensities across image regions to estimate camera motion and scene structure. Instead of first identifying and describing distinct keypoints and then matching them between frames, these methods directly minimize a "photometric error." This error quantifies the difference in pixel intensities of corresponding points as the camera moves.

4.2 Benchmarking Datasets

To test the VIO methods we will run the methods on two different datasets. The widely used EuRoC MAV dataset [2], and our own recorded AUDrone dataset.

4.2.1 EuRoC dataset

The EuRoC dataset is a collection of 11 different test drone flights, with varying movements ranging in difficulty to map. The easy data to map is typically linear foreward movement in a bright invironment, while difficult is non-linear movement in all directions (sideways and backwards) and in low lighting. These different scenarios are to test the VIO-method's robustness to different real-world conditions. The dataset includes stereo video and IMU data and the recorded 'ground truth'. The ground truth data is recorded with a motion capture system and a laser tracker.

The EuRoC drone uses a Aptina MT9V034 global shutter camera at 20 fps, and a ADIS16448 IMU at 200 Hz. [2]

Figure 1: EuRoC dataset example image.

4.2.2 AUDrone dataset

We wanted to make our own dataset, so we ...

The drone we are using is borrowed from the AU drone lab, specificaly it is the PX4 Autonomy Developer Kit drone. It features dual fisheye cameras and an IMU sensor. One of the cameras is for high quality video and streaming, the other for VIO tracking, so we will only use the tracking camera. The tracking camera is a On-Semi AR0144, which is specifically designed for industrial solutions, in our case the only important specification is that it is using a global shutter. If we were to use the high quality camera with rolling shutter it would produce a lot of tracking problems for the VIO algorithm. The IMU is time synchronized with the camera, so that when logging data from the drone we get a precice and syncronized timestamp of the tracking images and IMU datapoints. The integrated MEMS IMU is running at 130 Hz and the camera at 29.97 fps. In the chapter 'system setup' we explain how we calibrated the camera and IMU.

Example from dataset seen on figure 2.

Figure 2: AUDrone dataset example image.

Ground truth acquisition?

5 Methodology

5.1 Research Approach

We adopt a quantitative, comparative research design to systematically evaluate the performance of various monocular VIO algorithms. This approach is essential as it allows for direct comparison of algorithm capabilities, identifying their relative strengths, weaknesses, and suitability for different operational environments, contributing to an informed selection for future applications or work.

5.1.1 Setup

All experiments were conducted on a single laptop equipped with a ryzen 9 5900HS CPU, 32GB RAM, running a virtual machine with either; Ubuntu 16.04, 18.04, 20.04 or 22.04, LTS versions. Further setup information is specified in the 'system setup' chapter. But the tests were conducted on a single setup to remove as much inconsistency and bias as possible. Each VIO algorithm was executed using its provided ROS wrapper or standalone executable. Default parameters for EuRoC were employed for each algorithm, to ensure a fair comparison based on their out-of-the-box performance. Some specific parameters were adjusted when testing the PX4 drone (e.g., IMU noise, camera distortion parameters, resolution and so on), these adjustments can all be found in the calibration files here: audrone_config.

5.1.2 Evaluation metrics

The performance of each VIO method was quantitatively assessed using established metrics commonly employed in the visual odometry and SLAM literature. The primary evaluation metrics are:

- Absolute Trajectory Error (ATE): This metric quantifies the global consistency of the estimated trajectory by comparing the estimated camera poses directly to the ground truth poses after a rigid-body transformation (translation, rotation, and scale) has been applied to align the trajectories. We opted for the Mean Absolute Error (MAE) instead of the more common Root Mean Square Error (RMSE) when calculating the ATE score. This decision was made to minimize the impact of extreme values. To get the ATE score, we calculate the Mean Absolute Error of the translational components, providing a single scalar value representing the overall global drift. [3]
- Computational Performance: The average processing usage [%] and memory usage [%] were recorded for each method to evaluate its real-time capabilities and overall efficiency.

These two measurements should ensure that an optimal VIO method can be selected for a given drone, where accuracy and computational requirements are typically trade-offs.

5.2 Selection of VIO methods

To ensure a comprehensive evaluation, we aimed for a wide selection of VIO methods. Our criteria included choosing a mix of established and actively developed techniques, with publicly available source code and varied architectural designs (e.g., filter-based, optimization-based, feature-based, direct methods). Additionally, we specifically incorporated methods with SLAM capabilities to assess their potential impact on performance. Overall the methods were loosely choosen based on this survey VO-Survey, which not only links to a lot of visual odometry methods, but also gives a quick overview of their last activity, who made it, which license it has and its features.

We went with the following 9 VIO methods for our testing. Based on both the advice of our advisor and generally, which had a better combination of features and were considered more "active" or still worked on.

- ORB-SLAM3 [4] A versatile, well-regarded, and widely referenced open-source system, ORB-SLAM3 provides SLAM capabilities for monocular, stereo, and RGB-D cameras, and also includes visual-inertial functionality. Although the author is no longer actively developing it, the method remains community-driven.

- Vins-Mono [5] VINS-Mono is an optimization-based monocular visual-inertial system. It is primarily designed for state estimation and feedback control of autonomous drones and is also capable of providing accurate localization for augmented reality (AR) applications.
- Vins-Fusion [6] VINS-Fusion is an extension of VINS-Mono. Tt's used for vision-aided autonomy, SLAM, and AR, offering more flexibility in sensor configurations.
- SVO-PRO [7] SVO-PRO use a semi-direct approach, leveraging both pixel intensities and features for motion estimation. SVO-PRO supports various camera types in monocular or stereo setups and includes visual-inertial odometry and SLAM capabilities with loop closure. It is known for requiring relatively low computational resources, making it an interesting option.
- Maplab 2.0 [8] Maplab 2.0 focuses on multi-robot SLAM, enabling many diverse robots with various sensors to collaboratively map large areas. A key feature is its integrated VIO method, ROVIOLI, which we though would be interesting to include.
- VI-DSO [9] VI-DSO is a VIO approach that jointly estimates camera poses and sparse scene geometry by minimizing photometric and IMU measurement errors. Unlike keypoint-based systems, it directly minimizes a photometric error, allowing it to track pixels with sufficient intensity gradients. Could potentially have higher accuracy in certain environments compared to feature-based methods.
- OKVIS2 [10] OKVIS2 is an extension of OKVIS, a lightweight optimization-based VIO method. OKVIS2 includes features like loop closures and semantic segmentation for filtering dynamic objects, enhancing its robustness and mapping capabilities. It's still being actively developed.
- R-VIO [11] R-VIO is an efficient, lightweight, sliding-window filtering-based VIO. It's specifically designed to work well on monocular camera and a single IMU platforms.
- KIMERA [12] Kimera is a popular VIO and SLAM method, currently under active development. It achieves robust pose estimation by tightly integrating visual and inertial data, compatible with both monocular and stereo camera setups. Kimera also incorporates a more complex Simultaneous Localization and Mapping (SLAM) system, enabling advanced features such as 3D mesh reconstruction and semantic labeling.

6 System Setup and Configuration

Each Visual-Inertial Odometry (VIO) system was configured according to its official documentation and should all have ready made configuration files to run on the EuRoC MAV dataset. Where necessary, minimal changes were made to ensure consistency and comparability across systems. A notable example is the enforcement of headless execution; not all systems provided built-in support to run without a graphical user interface (GUI), which could otherwise skew performance metrics such as CPU and memory usage. In such cases, we manually modified the source code to disable external visualizers like RViz or Pangolin. Core algorithmic parameters such as the number of extracted features, loop closure behavior, and threading models were left at their default values to maintain a fair comparison and avoid introducing bias due to uneven tuning or system-specific optimizations.

6.1 Configuration of VIO systems

ORBSLAM3_ROS2 Running on Ubuntu 22.04, was installed with pangolin (commit 122bb3e), openCV 4.4.0, eigen3 3.4.0 and ros2 humble, several included custom thirdparty tools and is build with colcon. It utilizes the vocabulary file and yaml file included inside of the ORBSLAM3 project. Now the original version of this ROS2 wrapper doensn´t come with a mono inertial mode, but comes with a mono and a stereo inertial mode. So a modification had to be made to the source code to enable us to run with a mono inertial mode instead, hence a new mono inertial mode was added by combining similar code logic and structure to that of the stereo and the mono modes and can be found here Mono inertial code.

In this implementation, the MonoInertialNode constructor sets up the core functionality of the node. It subscribes to a camera topic and calls GrabImage() each time an image is received. Similarly, it subscribes to an IMU topic and triggers GrabImu() for every incoming IMU message. The constructor also initializes three key components: first, it inherits from a ROS 2 node; second, it receives a pointer to the ORB-SLAM3 system for tracking; and third, it spawns a dedicated thread that runs SyncWithImu() to synchronize image and IMU data for accurate visual-inertial odometry.

The GrabImage() and GrabImu() functions are responsible for buffering incoming image and IMU messages, respectively. Each function locks a shared mutex to ensure thread safety, adds the incoming data to its corresponding queue, and performs cleanup to prevent buffer overflows. Additional logic is included to aid in debugging and to verify timestamp consistency. GrabImage() also attempts to find the closest matching IMU timestamp for logging purposes, while GrabImu() logs IMU timestamps to file for further analysis.

The SyncWithImu() function runs in a separate thread and continuously checks for synchronized image and IMU data. It first locks access to the buffers, retrieves the next available image, and filters the IMU queue to include only messages within a certain time window around the image timestamp. Old or future-dated IMU data is discarded. The valid IMU readings are converted into a std::vector¡ORB_SLAM3::IMU::Point¿, which, along with the image, is passed into ORB-SLAM3's TrackMonocular() function for processing. This synchronization ensures robust tracking while avoiding issues such as NaN values due to misaligned data, which was a huge issue with early testing of this code on the AUDRONE, when we didn´t have proper configuration files and very optimized bag recording.

Lastly we have a separate node by the name of imu_combiner_node, which is only necessary cause we use the AUDRONE´s px4 imu, known as IMU_apps, that publishes two different topics '/fmu/out/sensor_combined' and '/fmu/out/vehicle_attitude', that have our needed imu data. A combination of both is then required to be fed into our orbslam3 instance, we instead let ORBSLAM3 subscribe to this topic node instead, which combines the information from the two other topics. While the functionality is quite simple, simply subscribing to each topic and combining it into a new topic, which it then publishes, there are a bit of extra functionality in the node. One of these additional functionalities is, that the node also serves as a logger for logging and analyzing several interesting imu topics like imu_combined, imu0 and more. This was done to see if some of our nan errors were related to bad timestamps, duplicate messages, empty imu messages or other related issues, which had to be confirmed by checking the consistency of the imu's messages.

Running the program is done using the command:

```
ros2 run orbslam3 mono-inertial /home/jeppe/Desktop/Bachelor/colcon_ws/src/
    orbslam3_ros2/vocabulary/ORBvoc.txt  /home/jeppe/Desktop/Bachelor/ORB_SLAM3/
    Examples/Monocular-Inertial/EuRoC.yaml --ros-args --remap camera:=/cam0/
    image_raw --remap imu:=/imu0
```

VINS-MONO Running on ubuntu 16.04 with ROS Kinetic including some additional ros packages like ros-kinetic-cv-bridge ros-kinetic-tf ros-kinetic-message-filters ros-kinetic-image-transport, build with catkin.

Running the program is done using the command:

```
Terminal 1: roslaunch vins_estimator euroc.launch
Terminal 2: roslaunch vins_estimator vins_rviz.launch
Terminal 3: rosbag play /home/jeppe/Desktop/Vins-Mono/MH_01_easy.bag
```

VINS-FUSION Running on ubuntu 16.04 with ROS Kinetic, ceres solver 2.20 CMake 3.16, Eigen 3.3 and is build using catkin. Running the program is done using the command:

```
Terminal 1: roslaunch vins vins_rviz.launch
Terminal 2: rosrun vins vins_node /home/jeppe/Desktop/Vins-Fusion/catkin_ws/src/VINS-Fusi
Terminal 3: rosbag play /home/jeppe/Desktop/Vins-Mono/MH_01_easy.bag
```

SVO-PRO (also known as rpg_svo_pro_open) Running on ubuntu 18.04 with ROS Melodic, catkin tools, vcstools, ceres solver, Global map feature utiilizng iSAM2 (this should be able to be skipped as it was a lot of trouble to get it to build) and is build using catkin. Running the program is done using the command:

```
Terminal 1: roscore
Terminal 2: roslaunch svo_ros euroc_vio_mono.launch
Terminal 3: rosbag play MH_03_medium.bag
```

R-VIO Running on ubuntu 16.04 with ROS Kinetic, utilizing eigen 3.1.0, openCV 3.3.1, tf, sensor_msgs, geometry_msgs, nav_msgs, cv_bridge, eigen_conversions and is build using catkin. Running the program is done using the command:

```
Terminal 1: roscore
Terminal 2: roslaunch rvio euroc.launch
Terminal 3: rosbag play /home/jeppe/Desktop/Vins-Mono/MH_01_easy.bag /cam0/image_raw:=/ca
```

VI-Stereo-DSO Running on ubuntu 20.04 with Suitesparse, eigen3, opencv,pangolin (commit 122bb3e), ziplib. Build using cmake + make. Running the program is done using the command:

```
Terminal 1: ./run_euroc.bash '$dataset_path'
```

MAPLAB Running on Ubuntu 18.04, with ROS-Melodic, has a list of dependencies far longer than would be worth mentioning, what is a smart idea, is setting up the ccache to cache intermediate build files while running the build process. The actual make of maplab will build some 150 dependencies with catkin, which is nice to not have to do, but also be mindfull of your -j($nproc) flag value, some of them are extremely heavy on ram requirements.Similarily to others it should go mostly smoothly, but if not analyze the build error, you might have missed a sudo apt get. Running the program is done using the command:

```
Terminal 1: roscore
Terminal 2: rosrun rovioli tutorial_euroc "$OUTPUT_MAP" "$BAG_PATH"
```

OKVIS2 Running on ubuntu 20.04, with ROS-Noetic. Has google-glog + gflags, blas and lapack, eigen3, suitesparse and cxsparse, boost, OpenCV via libopencv-dev, LibTorch. Is fetched with the git clone –recurse-submodules, then initializing them and lastly normal cmake + make. Running the program is done using the command:

```
Terminal 1:./okvis_app_synchronous <path/to/euroc.yaml> <path/to/MH_01_easy/mav0/>
```

KIMERA Running on ubuntu 20.04, with ROS-Noetic. Has GTSAM's Optional dependencies. Is build using catkin and utilizes wstool´s to install dependencies from the rosinstall file in the repo, note that you need a setup git on your vm or manually go and change all the links to https instead for this to work. Outside of the mentioned dependencies, the build process will most likely need several additional packages that it for some reason doesn´t have in the wstool´s or fails to get or mention. Keep building, look at the errors and sudo apt get whatever is missing. Running the program is done using the command:

```
Terminal 1: roscore
Terminal 2: roslaunch kimera_vio_ros kimera_vio_ros_euroc.launch
Terminal 3: rosbag play --clock "$bagfile"
```

6.2 Data Preparation and Preprocessing

The dataset used for evaluation needed to include groundtruth trajectories and be available in three formats: ROS 1 bags, ROS 2 bags, and EuRoC-MAV format. The EuRoC-MAV dataset proved ideal, as it not only came in both ROS 1 and EuRoC-native formats but also included groundtruth data embedded in each bag. Given its widespread use and support among VIO systems, it was a natural choice.

Getting the datasets were easy, simply going to euroc[2] website there is 22 download links, 1 for each dataset and format. Getting all 22 requires a total download of 37.5GB.

However, one final preparation step remained: converting the dataset into ROS 2 format for compatibility with the ORB-SLAM3 ROS 2 wrapper. For this, the rosbags Python tool was used. This utility enables straightforward conversion from ROS 1 to ROS 2 bag formats with a simple command:

```
rosbags-convert --src path/to/ros1bag.bag --dst path/to/ros2bag
```

After conversion there should now be all three different formats of the dataset, totalling 55.9GB of data.

Outside of the EuRoC-MAC dataset, we also wanted to collect our own dataset, with the available AU-DRONE, to see how easy it would be to utilize and optimize the VIO systems for a more "custom" and unfamiliar dataset. The actual preparation and preprocessing for that dataset also followed a similar structure to the EuRoC-MAV dataset, however differences did occur, which we will describe next.

6.3 AU Drone Setup and Calibration

For the data collection we were allowed to borrow a drone, and test it at an AU lab. The drone is running PX4, a highly configurable and open source flight control software. The drone itself is a dev kit from PX4, described earlier in chapter 4. First step to making the dataset is to calibrate the drone. Accurate calibration is essential for the correct operation of the various VIO methods. This calibration establishes key parameters, including camera distortion coefficients, IMU noise characteristics, and the extrinsic transformation between the camera and IMU. We performed this calibration using the open-source tool "kalibr". Kalibr was chosen partly because it is also used by several of the VIO systems we intend to test, such as ORBSLAM3 and OKVIS2. For the calibration procedure, we utilized a printed aprilgrid target, which provides robust and easily detectable feature points necessary for the calibration software. For accurate calibration, it is essential to excite all axes of rotation and translation during recording. We took this requirement into account when recording the calibration data. Flight data from the drone could be acquired in two ways: as a ROS bag with camera and IMU topics, or as raw data logs. We chose to log the raw data for increased flexibility, allowing conversion to a ROS bag later if needed. To access the drone we used the "Android Debug Bridge" shell. When connected we could start a log using the command:

```
Terminal1: adb shell
Terminal2: voxl-logger --preset_odometry --time *
```

This command logs all internal camera and IMU data for a set amount of time in seconds. When the recording is done we transfered the data with the following two commands, one for camera data and one for the IMU data:

```
Terminal1: adb pull /data/voxl-logger/log000*/run/mpa/tracking C:\Users\*\Desktop\
Terminal2: adb pull /data/voxl-logger/log000*/run/mpa/imu_apps C:\Users\*\Desktop\
```

The resulting data could be converted to a ros bag using a convertion script from kalibr.

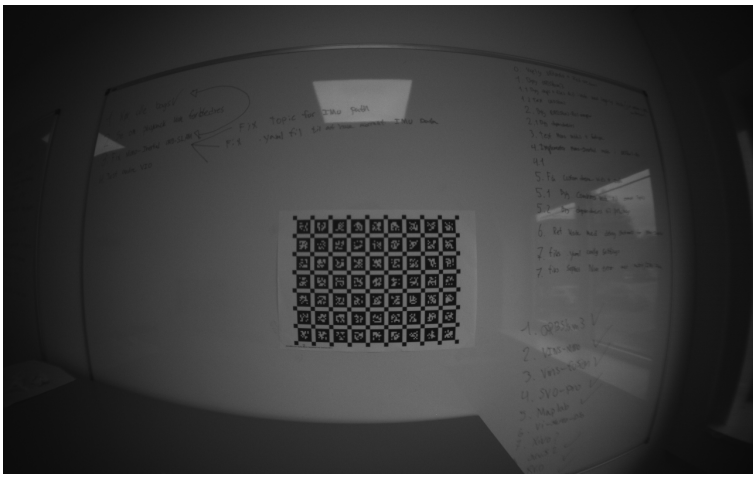
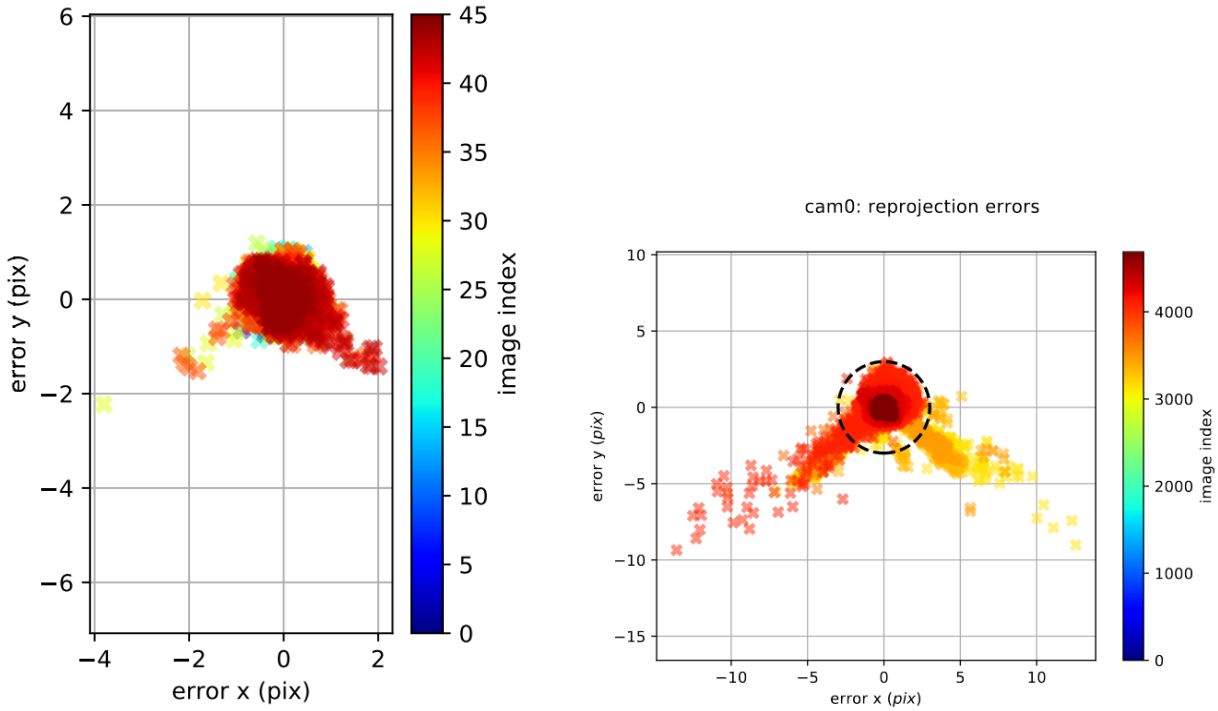


Figure 3: PX4 drone calibration ros bag example image.

The VIO methods we want to test are all either compatible with a ros bag or the EuRoC MAV dataset format, so the data from the drone needed to be converted. So we developed a custom script for data conversion from the px4 MPA log format to the EuRoC MAV format. The script can be found with a dedicated wiki page on our github here: voxl-mpa_to_euroc.

Running the Kalibr calibration with the ROS bag was straightforward. We started with the "multiple camera calibration" for the camera intrinsics, and then used the "camera IMU calibration" to get the IMU intrinsics and the camera-IMU extrinsics. [13] The final calibration file, containing all the necessary parameters derived from the calibration process, can be seen in the attachments. Figure 4 illustrates the calibration errors encountered during the process. Overall, the result is pretty good, indicating a successful calibration with only a few outliers.



(a) Camera calibration error.

(b) IMU calibration error.

Figure 4: PX4 drone calibration error results.

Each VIO method requires a calibration YAML file, but since their formats vary slightly, we had to manually input the calibration parameters into each VIO method's configuration files. You can find the final configuration files we used for testing here: audrone_config.

Insert picture of the drone lab and specify its location.

7 Testing

To test the different VIO methods, we will first look at a controlled and widely used dataset "EuRoC-MAV"[2], which contains a set of 11 drone flights with varying combinations of slow startup, slow movement, bright lighting, fast startup, fast movement, and dark lighting. The datasets are made based on two very different scenarios, one being a "Machine Hall" with lots of industrial equiptment, big structures, and open space. The other scenario "Vicon Room" is a small room with loads of small items, structures, and furniture. The data set also contains a ground-truth evaluation with a combination of Vicon motion capture system (6D pose), a Leica MS50 laser tracker (3D position) and a Leica MS50 3D structure scan, making the evaluation process extremely accurate and easy.

7.1 AU drone

At last we've conducted our own test in the AU drone lab. The drone data was exported into a ROS1 bag and we captured the drone flight with a PX4 autonomy developer kit.

The use of the AU drone bag will generally follow and be added on to the remaining 11 bags. But there are some extra steps that was necessarry for utilizing the drones bag for testing out our VIO programs. The programs are already very familar with the EuRoC-MAV dataset, which means all of them come with a form of YAML file already describing relevant information about the drone, camera and IMU calibration as well as relative position of any of the things in relation to each other, like the TBC: Matrix. While for most of these programs their own version of such a YAML file has been choosen, as it must have been the most suitable one without further extensive research into every single program and the dataset, to potentially make slightly better altered versions. Not all of the programs share the same structures for how such YAML files are written, so faith was given to the authors of each of the programs, that they had made a sufficiently good enough file.

The issue now was, to find a way for us to utilize the calibration metrics we calculated with tests and convert all of that into about 11 different yaml files and configurations. While certain characteristics were for the most part kept the same as their EuRoC-MAV yaml versions, like internal program functionality, all of the other information had to be tailored to our specific drone setup.

Alongside this another program had to be written, which combines PX4 rostopics from the AUDRONE at runtime, to spit out a combined IMU topic that could be fed to most of the programs.

7.2 Evaluation Metrics

All of the tests have been done on a variety of virtual machines in VMware Workstation, running Ubuntu versions 16.04, 18.04, 20.04 and 22.04. Each virtual machine was given 16GB of ram and 12 virtual cores running on an Asus Rog Zephyrus g14, with a ryzen 9 5900HS CPU. For the testing evaluation metrics, we have performance and accuracy, where performance is both cpu and memory usage during the tests and accuracy is the AME of the results compared to the EuRoC-MAV[2] groundtruths.

7.3 Testing Procedure and Reproducibility

The testing procedure is simple, run the software and all its dependencies as described in their Github repositories, monitor its usage with a logger script and close everything after a successfull run. Nearly all scripts follow this format, however some require special nuances and more attention from the user.

7.3.1 Automation of programs on all 11 bags

The first part of the testing process was automating the running of the programs, logging and retrying on failed attempts with scripts. So for this the scripts run_all_euroc_"programname".sh, log_resources_"programname".sh and run_all_AUDRONE_"programname".sh, were made, which can be found here: Program and log Scripts

The functionality of the scripts are quite simple, first off each logging script, simply uses in built linux commands like top to to get and then echo out the current memory and cpu usage in percentage into a log

file with an appropriate name.

The runner scripts, usually starts out with some form of path setups to logger scripts, the program specific path on the vm, the path to the specific bag versions (ros1, ros2 or maveurocformat), that the program needs to run it on and any other relevant or individual paths for that program.

This is followed up by a quick sourcing of the setup.bash file, that each program usually generates after succesfully building it. Afterwards it has some form of a launch command that is specified on each programs individual github repository. Lastly the whole "running" part of the script happens. For all the folders, that contains our bags, it launches both the program command, then the logger command and the bags and takes note of each of their processor id´s, it then monitors for a successfull exit of the program to then determine when to either move the generated trajectory and logging files or whether to close down all its stored processor ids and retry running it again.

As a sidenote, the way retrying is caught is usually determined by whether a crash signal was sent by the program or a trajectory file was generated by it. However all programs work quite differently so sometimes program specific solutions had to be made to retrieve a similar result.

7.3.2 Automation of trajectory data conversion

After succesfully generating trajectories for all Programs, came the process of confroming all the data to a specific format known as "TUM". This step is vital as it is required for the evaluation program we have choosen, called EVO, but more about that in the following section.

So more automation scrips needed to be made, which could take care of the process of turning widely differing .csv files into our needed "TUM" text files.

The process is again very similar for all scripts, however differ a bit depending on each programs specific way of logging down trajectory information, some like to add a lot of additional information and some only the requide, there are even programs like ORBSLAM3, which has a feature to export the trajectory directly in that format.

The script convert_all_to_tum.py specifies and runs all the actual conversion script with the paths to find the related trajectory files and the specific names for each variant. The conversion script variants all follow the name convert_programname"_tum.py and inside of it, it specifies which columns to keep from the CSV and reorganizes it in the "TUM" format, which looks as such (timestamp, tx, ty, tz, qx, qy, qz, qw), where ty,tz,qx is the 3D translation in meters and the qy, qz, qw is the unit quaternion representing rotation, where qw is the scalar part. Those scripts can be found here: Conversion Scripts

7.3.3 Automation of Evo evaluation with the groundtruths

After collecting and subsequently converting the data, we lastly need to evaluate the data's accuracy in comparison to our groundtruths using the EVO program[14].

For this a singular automation script was created with the name Evaluate_All_Program_On_Datasets.py. The script sets up two different structures, our first structure is our dataset structure, which simply tells us which folder names that our script is looking for (mh-01..., v1-01... etc). The next is the methods structure, which specifies 4 things, name, folder, file and log_file. Name is simply the name of the method it will use to make the result easily connectable to the program. Folder specifies, in which of the three underlying folders ros1bags, ros2bags or eurocmavformat, should we look. File specifies which trajectory file name we are looking for and log_file specifies, which log file.

Then the script loops through and searches our whole repository for folders that match our specified methods and if any of the named ones are found, it evaluates those method´s trajectory files to the equivalent dataset´s groundtruth.

EVO then spits out a short result table of mae (we modified EVO to give this metric as well), max, mean, median, min, rmse, sse and std. These are all transferred alongside the log files information, where the average of cpu and ram percentage over all the points are calculated and all of these data points are put into

a single line in a csv file, by the name of evo_ape_all_results.csv. With that csv file, our whole testing phase is done and ready for comparison.

8 Results

8.1 Analytical comparison

So by utilizing the tool Evo[14], we got a plethora of results as discussed in the previous testing part, here we will solely focus on the MAE score and memory + cpu usage. If interested the full results, they can be found here: Full Results

8.2 Table: Accuracy

To compare the pose ATE scores we compiled the resulting values inte table 2. The highlighted scores are the most accurate.

Dataset \ VIO-method		ORB-SLAM3	Vins-Mono	Vins-Fusion	SVO-PRO	Maplab	VI-DSO	OKVIS2	RVIO	Kimera
EuRoC	MH_01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	MH_02									
	MH_03									
	MH_04									
	MH_05									
	V1_01									
	V1_02									
	V1_03									
	V2_01									
	V2_02									
	V2_03									
AU Drone	ex.1									
	ex.2									
	ex.3									
	ex.4									
Result										

Table 2: Benchmark results: Monucular VIO-methods measured accuracy (ATE score) on multiple different dataset collections. The best score (lowest) is highlighted for each dataset. The results row shows the average accuracy across all datasets.

While the results of the programs are quite widespread in overall performance, there are some clear winners in terms of overall performance on all 11 datasets. First off the three best programs in overall accuracy has to be OKVIS2 and MAPLAB and VINS-MONO. The reason why, is that all of these programs in fact played the entirety of the dataset at default settings with absolutely no external help and got overall great results. Meanwhile the remaining programs needed some extra helps, in total it goes as follows. The programs that needed help with 4 datasets were kimera and still performed badly even after extensive help. The programs that needed help with 3 datasets were ORBSLAM3, RVIO and SVO-PRO with varying amounts, but also varying results. The programs that needed help with 2 datasets were VI-Stereo-DSO with quite some help and still couldnt really dial them in for great results on those outliers. The programs that needed help with 1 dataset were Vins-Fusion, which was very minor help and an overall admirable performance.

Now what does help in this case mean ? Well simply, there were a ton of irregularities, first we tried retrying datasets over and over again to see if we would get better result with just another try. This however rarely helped any of the outliers, as they got MAE scores in the 50´s to 10.000´s!

Some of the ways that we actually managed to get every dataset to run for every program and get a somewhat reasonable MAE score of (1m or less), was playing with two simple factors, which is playback

rate and skipping flag, for initialization flight.

The playback rate is simple, we simply slow down the rate at which the bag is being played at, which helps the programs overall process less information for each frame and better maps and calculates the information.

The skipping flag also denoted as (-s), simply tells the bag to skip that amount of seconds into the recording, completely ignoring all of the initialization of being stationary, flying a little and landing again. For specific programs, like RVIO this is actually an incredibly big problem for some of the recordings, but not all. There isn´t a set amount of time that bags needed to be skipped to make em work, but generally for MH-01 and 02, it was around the 25-35s mark. Where as with the MH-03 and 05 it was anything between 9-15s. Even some of the V1 and v2 bags also benefitted from skipping anywhere from 1-4s.

8.3 Table: CPU/Memory Usage

To compare the VIO methods we also have to look at computational and memory usage. Table 3 is expanded to include both CPU and memory(RAM) usage. Both of these metrics are in measured in units of [%] of the maximum available resource. The CPU metric is average % usage of 12 virtual cores on a ryzen 9 5900HS and the RAM metric is average % usage of 16GB of RAM. A lower score is better, indicating a more efficient algorithm.

Dataset / VIO-method		ORB-SLAM3	Vins-Mono	Vins-Fusion	SVO-PRO	Maplab	VI-DSO	OKVIS2	RVIO	Kimera
EuRoC	MH_01 CPU	00%	00%	00%	00%	00%	00%	00%	00%	00%
	RAM	00%	00%	00%	00%	00%	00%	00%	00%	00%
	MH_02 CPU									
	RAM									
	MH_03 CPU									
	RAM									
	MH_04 CPU									
	RAM									
	MH_05 CPU									
	RAM									
	V1_01 CPU									
	RAM									
	V1_02 CPU									
	RAM									
	V1_03 CPU									
	RAM									
	V2_01 CPU									
	RAM									
	V2_02 CPU									
	RAM									
	V2_03 CPU									
	RAM									
AU Drone	ex.1 CPU									
	RAM									
	ex.2 CPU									
	RAM									
	ex.3 CPU									
	RAM									
	ex.4 CPU									
	RAM									
Result	CPU									
	RAM									

Table 3: Benchmark results: Monucular VIO-methods measured CPU and memory usage on multiple different dataset collections. The best score (lowest) is highlighted for each dataset. The results row shows the average usage across all datasets.

8.4 Graphic comparison

Matlab graphs

9 Discussion

9.1 Key Findings

The most accurate VIO method is ... while the most efficient is ... . For these applications you should use ... method while in this application you should use ... method.

9.2 Strengths and Weaknesses of Evaluated VIOs

If it weren´t

9.3 Challenges Encountered During Research

Alot of the VIO systems suffers from being a bit deprecated and old, while several also suffers from having inconsistent or terrible installation steps and or explanations. Instead of detailing the installation process of every single VIO system, we will list a bunch of the most common faults and tripping points that happens:

Incorrect versions of third party dependencies: By far the most annoying error to fix, cause while the solution is simple, it isn´t easy to find out if this is the cause of your problem or something different, as you will usually be hit with hundreds of errors and deprecated functions. Your best bet is to try and find out if a post has been made specifying it to be the case, as the only fix is to either go into what ever thirdparty dependencies folders and git checkout a commit that was within the timerange of the vio programs life cycle, here google and AI chats can really help you narrow down timeframes quite a lot faster than manually guessing.

Cmakelists errors: Common problem, that can be easy to fix once you start learning more about the cmakeslist file, but the error types aren´t as easily understandable and easy to fix as other errors. Ususally you need to manually add in some form of include dir or remove them, because either you forgot to sudo make install or the automated install process that the vio program ran didn´t manage to do it properly either.

Cmake wrong version: Either Cmake is updated on your system, the vm you are running or it was just not run with such a version available at the time the program was made. These errors are problematic, leading either to trying to reinstall a whole new version of Cmake, which is a huge issue if you have several different VIO systems on one computer or VM. The best advice for tackling these is start with a fresh VM of exactly the version that the guide used or specified and then hope it has the cmake version you need and if not downgrade/upgrade it.

Deprecated or newer versions of functions and their use: These are quite common, program relies on using some functions that have now been deprecated and you need to manually change em to the updated ones, they are quite easily understandable and therefore easily fixable for the most part.

Another huge problem that one will encounter with the VIO systems, is both their error logs and their crash reports. Generally the programs will say tracking lost, map lost, restarting and more, these might sound good on the surface, but since there is no specifying exactly what it is that is causing the issue, it will be hard to realize and alleviate them.

Another one that was extremely frustrating with ORBSLAM3, was Sophus NAN errors. Especially one known in expandtheta(). Such and error caused months of work trying to debug exactly what was causing them and where in the code they were happening, later on whether it was related to the AUDRONE bags IMU readings, duplicated entries, too old or new IMU meassurements and much much more. Overall they are good things to now monitor for and clean with the use of the code, but it also meant that a lot of time was spend wastefully focusing on this aspect first, when it could have been saved for later, since a lot of NAN errors are attributed to bad bag recordings, drone imu and camera calibrations and their accompanying yaml files!

9.4 Limitations of the Study

This study is not an overall perfect study of the exact stability and performance of every program in a vacuum. Because of the nature of VM´s and the extensive testing pipeline and structure setup, there is no guarantee, that every program on this list will fare this well every time you try it, even if you replicate our exact settings. But given enough attention and tries, it should be possible to replicate similar results to ours at least once (a good/best case scenario). It also hasn´t in depth studied and played around with every intricate setting available for all nine of the VIO systems, while ORBSLAM3 is by far the most tinkered with program, even here it would be beneficial to try and test every single bag with varying program settings.

9.5 Lessons Learned

While our results showcase the overall expected performance and accuracy one can get as a relative newcomer with just minimal setup, knowledge and experience, it sadly falls short of dialing in every single VIO method intrinsically, both with their internal parameters, but also additional feature testing and relative dataset parameters. Had our study been more thorough and had significantly more time, it would most likely have been possible to try and make this whole testing setup one hundred percent automated. Starting up individual virtual machines, logging in, prepping the shared data folder and running the tests scripts. From there it would have been possible to add both more extensive testing of all manners of these parameters, like lets use ORBSLAM3 as an example. We could test parameters like, varying the playback rate of the bag from 0.1-1.0, skipping anywhere from plus or minus 5 seconds from a determined standstill point in each dataset. Trying to get at least 5 succesfull results and only allow it to 10 failure attempts due to crashes and then lastly we could try and mess around with the orb extractor intrinsic features of orbslam3, like ORBextractor.nFeatures: 1000 (could range 100-2000) ORBextractor.scaleFactor: 1.2 (could range 0.8-1.6 ORBextractor.nLevels: 8 (could range 6-12 ORBextractor.iniThFAST: 20 (could range 15-25) ORBextractor.minThFAST: 7 (could range 5-9)

While this would be the ideal ultimate testing scenario, even if we limited it to three different values for all of the categories and wanted to try lets say just 5 times at each setting, you are looking at a total of $3^7 = 2.187 combinations x 5 runs for a total of 10.935 runs of every single dataset, with an average length of about 2 minutes that would b$

10 Conclusion

References

[1] OpenCV. *OpenCV - Feature Detection and Description*. Last visited: 21-05-2025. https://docs.opencv.org/4.x/db/d27/tutorial_py_table_of_contents_feature2d.html.

[2] M. Burri - J. Nikolic - P. Gohl - T. Schneider - J. Rehder - S. Omari - M. Achtelik - R. Siegwart. *The EuRoC micro aerial vehicle datasets, International Journal of Robotic Research*. Last visited: 04-02-2025. https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets.

[3] Zichao Zhang - Davide Scaramuzza. *A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry*. Last visited: 20-05-2025. https://rpg.ifi.uzh.ch/docs/IROS18_Zhang.pdf.

[4] Carlos Campos et al. *ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial and Multi-Map SLAM*. Last visited: 19-05-2025. https://github.com/UZ-SLAMLab/ORB_SLAM3.

[5] Tong Qin - Peiliang Li - Zhenfei Yang - Shaojie Shen. *VINS-Mono A Robust and Versatile Monocular Visual-Inertial State Estimator*. Last visited: 19-05-2025. https://github.com/HKUST-Aerial-Robotics/VINS-Mono.

[6] Tong Qin - Shaozu Cao - Jie Pan - Peiliang Li - Shaojie Shen. *VINS-Fusion An optimization-based multi-sensor state estimator*. Last visited: 19-05-2025. https://github.com/HKUST-Aerial-Robotics/VINS-Fusion.

[7] Christian Forster - Zichao Zhang - Michael Gassner - Manuel Werlberger - Davide Scaramuzza. *SVO-PRO: Semidirect Visual Odometry for Monocular and Multicamera Systems*. Last visited: 19-05-2025. https://github.com/uzh-rpg/rpg_svo_pro_open.

[8] A. Cramariuc - L. Bernreiter - F. Tschopp - M. Fehr - V. Reijgwart - J. Nieto - R. Siegwart - C. Cadena. *maplab 2.0 – A Modular and Multi-Modal Mapping Framework*. Last visited: 19-05-2025. https://github.com/ethz-asl/maplab.

[9] J. Engel - V. Koltun - D. Cremers - Jiaming Sun. *VI-Stereo-DSO*. Last visited: 19-05-2025. https://github.com/RonaldSun/VI-Stereo-DSO.

[10] Stefan Leutenegger. *OKVIS2: Open Keyframe-based Visual-Inertial SLAM*. Last visited: 19-05-2025. https://github.com/smartroboticslab/okvis2.

[11] Huai Zheng - Huang Guoquan. *R-VIO: Robocentric visual-inertial odometry*. Last visited: 19-05-2025. https://github.com/rpng/R-VIO.

[12] Rosinol Antoni - Abate Marcus - Chang Yun - Carlone Luca. *Kimera: an Open-Source Library for Real-Time Metric-Semantic Localization and Mapping*. Last visited: 19-05-2025. https://github.com/MIT-SPARK/Kimera-VIO.

[13] Paul Furgale - Hannes Sommer - Jérôme Maye - Jörn Rehder - Thomas Schneider - Luc Oth. *Kalibr - gitHub*. Last visited: 22-05-2025. https://github.com/ethz-asl/kalibr.

[14] MichaelGrupp. *EVO*. Last visited: 04-02-2025. https://github.com/MichaelGrupp/evo.

11 Attachments

Raw VIO output pose data, VIO config files and data convertion scripts can be found on our GitHub: GitHub

11.1 PX4 developer kit - kalibr calibration results

```
Calibration results
===================
Normalized Residuals
----------------------------
Reprojection error (cam0):
  mean 0.34163014286935073,
  median 0.3084579533217657,
  std: 0.24238589339580188
Gyroscope error (imu0):
  mean 0.2187464073107878,
  median 0.17855148847126087,
  std: 0.20326054269390476
Accelerometer error (imu0):
  mean 0.24888541858128713,
  median 0.19178190055630998,
  std: 0.36532217570726927

Residuals
----------------------------
Reprojection error (cam0) [px]:
mean 0.34163014286935073,
median 0.3084579533217657,
std: 0.24238589339580188
Gyroscope error (imu0) [rad/s]:
mean 0.0024940927794448666,
median 0.002035800192698037,
std: 0.0023175267567190436
Accelerometer error (imu0) [m/s^2]:
mean 0.02837730379318959,
median 0.021866500999311134,
std: 0.041653136698517765

Transformation (cam0):
-----------------------
T_ci:  (imu0 to cam0):
[[ 0.00113902  0.99998275  0.00576216 -0.00101329]
 [-0.75731633 -0.00290039  0.65304178  0.01356732]
 [ 0.65304723 -0.0051076   0.75729996 -0.02326294]
 [ 0.          0.          0.          1.        ]]

T_ic:  (cam0 to imu0):
[[ 0.00113902 -0.75731633  0.65304723  0.02546771]
 [ 0.99998275 -0.00290039 -0.0051076   0.0009338 ]
 [ 0.00576216  0.65304178  0.75729996  0.00876284]
 [ 0.          0.          0.          1.        ]]

timeshift cam0 to imu0: [s] (t_imu = t_cam + shift)
0.005641694733126712
```

```
Gravity vector in target coords: [m/s^2]
[ 0.08638968 -9.80614797  0.02053661]


Calibration configuration
=========================

cam0
-----
  Camera model: pinhole
  Focal length: [576.2468107909041, 574.4047554715168]
  Principal point: [619.3982162142439, 396.7258120292723]
  Distortion model: radtan
  Distortion coefficients: [
    -0.3247272858176051,
    0.09935591920986149,
    0.00046215968702370514,
    0.001283410372345202]
  Type: aprilgrid
  Tags:
    Rows: 7
    Cols: 9
    Size: 0.029 [m]
    Spacing 0.00899 [m]



IMU configuration
=================

IMU0:
 ----------------------------
  Model: calibrated
  Update rate: 130.0
  Accelerometer:
    Noise density: 0.01
    Noise density (discrete): 0.1140175425099138
    Random walk: 0.001
  Gyroscope:
    Noise density: 0.001
    Noise density (discrete): 0.01140175425099138
    Random walk: 0.001
  T_ib (imu0 to imu0)
    [[1. 0. 0. 0.]
     [0. 1. 0. 0.]
     [0. 0. 1. 0.]
     [0. 0. 0. 1.]]
  time offset with respect to IMU0: 0.0 [s]
```

11.2 Project Proposal

AARHUS UNIVERSITY

Rev. 10/5-2023

Specifications of Bachelor project for BEng students at Aarhus University / Dept. of Electrical and Computer Engineering

Please fill out the form and send it to Rasmus Nielsen (rani@ece.au.dk). When is has been approved by the Head of degree programs it will be published on Brightspace. A supervisor will be allocated at the end of the semester.

Date	24/11/2024											
Project title	Comparison of Monocular Visual(-Inertial) Odometry											
The project applies to at least 2 students within: (Multiple selections are welcome)	Electronic Engineering			Software Technology			Electrical Power Technology			Healthcare Technology		
	x			x								
Who has initiated the project? (Mark with an X in either Student, Company, or ECE Staff)	Student(s)	Name 1:			Name 2:			Name 3:				
		Mads Evander Jensen			Jeppe Emil Smedegaard Pape							
		Student number 1:			Student number 2:			Student number 3:				
		202107547			202008596							
	Mark with X	E	SW	EE	ST	E	SW	EE	ST	E	SW	EE	ST
		X					X						
	Company	Company name		Contact name		Email		Phone				
	ECE Staff	Name				Email						
	x	Andriy Sarabakha				andriy@ece.au.dk						
Requested supervisor (need only be filled out if relevant) – May be overruled by ECE if necessary	The supervisor has been contacted and has indicated a willingness to supervise the project											
	Name of ECE staff				Email							
	Andriy Sarabakha				andriy@ece.au.dk							
Special demands to: – Equipment – Place – Confidentiality												

Questions about the *content* of bachelor projects can be directed to the head of degree programs:

- Electronic Engineering
- Software Technology
- Electrical Power Technology
- Healthcare Technology

Bachelor project learning goals

- Translate research results and scientific and technical knowledge into practice by using them in development projects and in the solving of technological problems.
- Search for, analyse and assess new knowledge within relevant areas.
- Develop new solutions.
- Use engineering theories and methods in a systematic manner.
- Assess and explain project results to engineers and other target groups, in speech as well as in writing.
- Consider how the project results can be applied socially, organisationally, environmentally, economically, ethically and in relation to sustainability and work environment.

Please describe the project in the field on the next page. Be sure to include:

- What is to be developed?
  - Due to the learning goals of Bachelor of Engineering, the bachelor project must be oriented towards development.
- Sufficient details to allow the head of degree programs to understand why the project is relevant for each specialization.
- What might the technologies involved be (software, hardware, processes)?
- If the project depends on external factors, e.g., access to equipment or data do you have a plan for proceeding if the equipment or data is delayed or unobtainable?
- Is the project realistic in scope pertaining to a bachelor project (time/money)?
  - If the project is large in scope with regards to time, consider stating a prioritized list of features to develop conforming to the MoSCoW method allowing agile development
- Illustrations and figures to illustrate the problem and the envisioned product or context.
- One or two references (e.g., scientific articles) if appropriate.

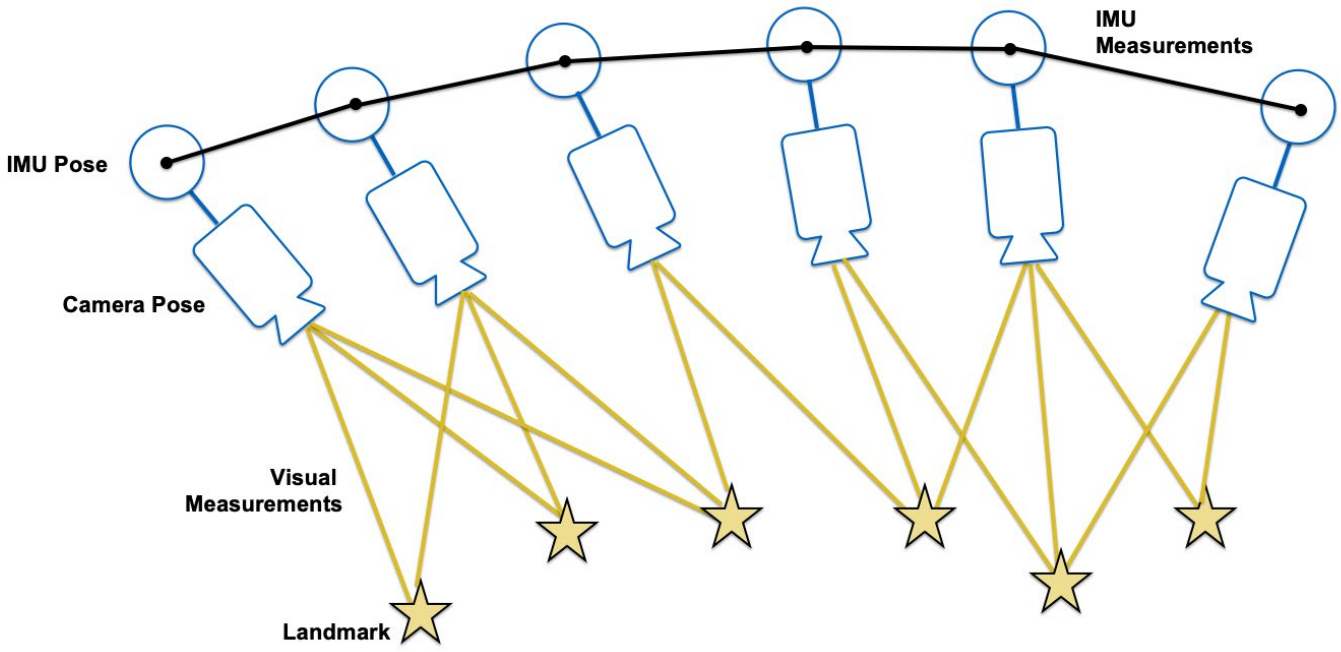
Project description

Objective:
The aim of this project is to compare several existing monocular visual and visual-inertial odometry methods for simultaneous localisation and mapping in an aerial robotics case study.

Motivation:
Monocular visual-inertial odometry estimates the position and orientation of the robot using camera and inertial measurement unit (IMU) sensor data. Camera-based state estimation is accurate during low-speed navigation. However, camera-based estimation faces challenges like motion blur and track loss at higher speeds. Also, a monocular camera-based estimation can estimate poses at an arbitrary scale. On the other hand, inertial navigation can handle high-speed navigation easily and estimate poses at a world scale. Combining the advantages of both types of sensor data is possible to achieve better accuracy. Visual-inertial odometry is currently applied to state estimation problems in a variety of domains, including autonomous robots.

Requirements:
- Good programming skills in C++.
- Experience in robot operating system (ROS).

References:
[1] J. Delmerico and D. Scaramuzza, "A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD, Australia, 2018, pp. 2502-2509, doi: 10.1109/ICRA.2018.8460664
[2] M. Servieres, V. Renaudin, A. Dupuis, and N. Antigny, "Visual and Visual-Inertial SLAM: State of the Art, Classification, and Experimental Benchmarking", *Journal of Sensors*, vol. 2021, 2021, doi: 10.1155/2021/2054828
[3] https://github.com/klintan/vo-survey



Comments from head of degree: