Introduction

Simultaneous Localization and Mapping (SLAM) allows a system to track its position while building a map of its surroundings—crucial for robots, drones, autonomous vehicles, and AR/VR devices operating without GPS. While many SLAM methods exist, their real-time practicality varies widely; some deliver excellent accuracy but demand heavy compute, while lighter approaches may struggle in complex environments.

This project focuses on evaluating how different SLAM algorithms perform under realistic conditions and identifying which ones are practical for real-time use on accessible hardware rather than only suitable for research setups. Benchmarks, configurations, and additional resources can be found in the GitHub Repo:

https://github.com/IGrilex/ASP

Experimental Setup

The goal of this evaluation is to see whether the tested SLAM systems can realistically operate in real time. The dataset runs at 10 FPS, so anything that can't keep up—whether due to high latency per frame or a big drop in output FPS—won't be usable in real scenarios like mobile robots or AR devices.

All experiments were run on a mid-range desktop PC, which is already more capable than most embedded platforms. So the rule of thumb here is simple: **if a SLAM algorithm struggles on this system, it's unlikely to run well on battery-powered or compact hardware**. Systems that max out the GPU just to stay afloat, sit at 100% CPU usage, or eat up large amounts of RAM are not practical outside controlled lab environments. Real robots still need computing space for navigation, safety checks, and other core tasks—not just SLAM.

In short, accuracy only matters if the system can deliver it efficiently. A SLAM method that can't keep pace with the camera stream on this PC won't suddenly become real-time on a smaller device, and that limits its usefulness for deployment.

Hardware

All experiments were executed on a standard consumer-grade workstation to approximate what is realistically available for robotics development. The system consists of:

- **CPU:** Intel Core i7-8700K, 6 cores / 12 threads @ 3.70 GHz
- **GPU:** NVIDIA GeForce RTX 4060 Ti with 8 GB VRAM
- **RAM:** 16 GB DDR4 @ 3467 MHz
- **Storage:** 2 TB Samsung 990 PRO SSD
- **Operating System:** Windows 10 Pro running WSL2 with Ubuntu 22.04 / 20.04
- **CUDA Toolkit:** 12.4 with driver runtime 13.0 supported
- **ROS Version:** ROS Noetic when required

Software Configuration

All tests were run on a Windows 10 Pro workstation, using WSL2 to provide a Linux environment required for SLAM workflows. Two Ubuntu distributions were used—20.04 and 22.04—depending on the dependency requirements of each algorithm. When a project recommended or required a containerized setup, desktop Docker was used to match the configuration described in its Repo.

GPU-accelerated methods were supported through the CUDA Toolkit (12.4) installed on the host Windows system, which allowed modern deep-learning-based SLAM frameworks to run without compatibility issues.

Benchmarks showing CPU and GPU usage at idle for both Ubuntu distributions are available in the GitHub Repo accompanying this project, providing a clearer picture of baseline resource consumption before SLAM execution.

Dataset & Preprocessing

All tests were performed using the **KITTI-360 dataset**: https://www.cvlibs.net/datasets/kitti-360/index.php

A large-scale real-world driving dataset covering **73.7 km**, with **~320,000 images** and **~100,000 laser scans**. It includes synchronized data from:

- Front-facing stereo cameras (≈90° FOV, ~60 cm baseline)
- Left/right fisheye cameras (≈180° FOV each)
- Velodyne HDL-64E spinning 3D LiDAR
- SICK LMS push-broom laser scanner
- GPS + IMU localization system

The dataset runs at **1408 × 376 resolution, 24-bit, 10 Hz**, which reflects a practical real-world frame rate and provides natural challenges including **motion blur**, **low light**, and **fast vehicle dynamics**—all relevant stress conditions for SLAM.

Two subsets were used:

- **Light** — only **test0** for slower algorithms
- **Full** — **test0, test1, test2, test3** for most evaluations

Dataset Setup & Preprocessing

Files were downloaded directly from:
https://www.cvlibs.net/datasets/kitti-360/download.php

Required downloads:

Section	Files	Size
2D data & labels	Test SLAM	~14 GB
3D data & labels	Test SLAM	~12 GB
Calibrations & Poses	Calibrations + Vehicle Poses	~9 MB

Setup steps:

1. Clone my repo.
2. Extract all archives from the kitti360 and place them into the /data.
3. Run:
4. python3 ./src/build_kitti360_test_maps.py

— This generates the .ply point-cloud ground-truth maps used for comparison.

No filtering, resizing, or image modification was applied. The goal was to evaluate SLAM performance **as-is** using real-world data without artificial preprocessing that might inflate results.

Evaluation Methodology

The evaluation is built around two main questions for each SLAM system:

1. How close is the estimated trajectory to ground truth?
2. Can it run near real time on my machine without maxing out CPU/GPU/RAM?

Each test sequence from KITTI-360 is processed in two phases:

Phase 1 – Runtime & System Load
For each sequence, the SLAM algorithm is run once over all frames in order. During this "online" run, the scripts measure:

- Total wall time for the sequence
- Average FPS = number_of_frames / wall_time (compared to the 10 Hz dataset rate)
- Number of frames for which the algorithm produced a valid pose
- Periodic system samples (CPU, RAM, GPU usage)

At the end, the run produces a TUM-format trajectory file (POSES_<drive>.txt) plus a machine usage log (MACHINE_USAGE_<drive>.txt). These are used to decide whether the method is realistically close to real time on my hardware.

Phase 2 – Accuracy Metrics (ATE & RPE)
In the second phase, the predicted trajectory is aligned and compared against KITTI-360 ground truth:

- Absolute Trajectory Error (ATE): after Sim(3) alignment (Umeyama), we compute RMSE, mean, median, and max position error, plus the estimated scale.
- Relative Pose Error (RPE): we compare local motion over short and longer windows (e.g., 0.5 s and 2.0 s) and report translational and rotational RMSE.

All of this—ATE, RPE, FPS, wall time, and summarized system usage, is written into per-sequence dashboard files, giving a compact overview of global accuracy, local consistency, speed, and computational load for each algorithm.

ORB-SLAM3 (Baseline)

ORB-SLAM3 is a classical feature-based SLAM system and one of the most widely used methods in the community, so it serves as the baseline in this project.

For implementation, I used the Python wrapper
Repo: https://github.com/xingruiyang/ORB-SLAM3-python

This keeps the whole project in Python and avoids dealing directly with a full C++ build, at the cost of some runtime overhead compared to the native implementation.

ORB-SLAM3 was tested in both monocular and stereo

Monocular:

- Using a KITTI-360 pinhole camera model (1408×376 @ 10 Hz) with **bf = 0** and **2000 ORB features per frame**.

Stereo:

- Using the calibrated KITTI-360 left/right pair with a **baseline of ≈0.594 m** (**bf ≈ 328.25**) and **3000 ORB features per frame**.

You can test the code by running the fallowing commands:

- Mono: python3 ./src/run_mono_ORBSlam3.py
- Stereo: python3 ./src/run_stereo_ORBSlam3.py

Installation on **WSL2 Ubuntu 22.04** was straightforward and problem-free. Performance was slightly below native C++ expectations, but both mono and stereo modes executed smoothly, with fast processing and clean trajectory/map outputs. In practice, stereo ORB-SLAM3 clearly outperformed monocular mode in terms of accuracy and map consistency.

MASt3R-SLAM

MASt3R-SLAM is a dense SLAM system built on the learned 3D-reconstruction prior MASt3R. Instead of relying on sparse keypoints, it predicts full "pointmaps" from images, matches them between frames to estimate camera motion and build a **dense 3D map**, with loop-closure and global optimization to keep poses and geometry consistent.

Repo: https://github.com/rmurai0610/MASt3R-SLAM

How I set it up

- I cloned the MASt3R-SLAM Repo (with submodules).
- I tried to follow the instructions on GitHub, which use a conda create + conda install … workflow — but that failed due to dependency issues on my system.
- Instead, I used the instructions from a community YouTube walkthrough (https://www.youtube.com/watch?v=TK8DK19o6YQ), but replaced **every** conda install … with pip install …, while still inside the conda environment. That was the only way I got the code running reliably.

My experience and problems

- The algorithm successfully produced dense point-cloud maps.

- However, performance was very slow: even after optimizations (using fp16 for the decoder and skipping frames — processing only every third frame), I measured roughly **1.4 FPS**, which is far below what's needed for real-time use.
- Because of this slow speed, I only tested on the "light" scenario (test0). Running the full dataset would take too long to be practical in my setup.

AirSLAM

This method is a visual SLAM system that combines a learning-based front end for feature detection (points + structural lines) with a classical optimization back end, this hybrid design helps it stay robust under challenging lighting or texture conditions.
It supports stereo (and optionally visual–inertial) input, does VO/VIO, map optimization (bundle adjustment, loop closure), and offers a relocalization pipeline for long-term use.

Repo: https://github.com/sair-lab/AirSLAM

Setup & How I Ran It

- Initially I tried to get it running natively on the Ubuntu 20.04 (so without a docker) but I got into a lot of dependencies errors.
- I used **Ubuntu 20.04 + Desktop Docker**, following the recommendation from the Repo, which worked reliably.

Inside Docker:

1. docker start air_slam (how I named my docker container)
2. docker exec -it air_slam /bin/bash (run every time you enter the docker)
3. cd /workspace
4. source devel/setup.bash
5. cd /ASP
6. Run preprocessing:
    - python3 ./src/make_data_4_AirSlam.py (this rearranges input images into the format expected by Air SLAM)
7. Run the algorithm:
    - python3 ./src/run_AirSlam.py

What I Observed

- AirSLAM successfully produces **dense point-cloud maps** and trajectories.
- However, I couldn't import AirSLAM as a library so I had to run it strictly via the provided script from the repo.
- I couldn't access the map as it outputs a .bin map.
- This was by far the best method as it produced almost GT level paths and it ran fast and didn't use to much CPU and GPU.

MAC-VO

MAC-VO is a recent learning-based **stereo visual odometry (VO)** system that uses a "metrics-aware covariance" approach: it estimates matching uncertainty for stereo keypoints and uses that to select reliable features and weight them during pose graph optimization.

Repo: https://github.com/MAC-VO/MAC-VO

Setup & execution

I installed MAC-VO using Docker. To run on my data, I used roughly the following commands:

- cd ~/ASP-SLAM
- docker run --gpus all -it --rm \

  -v "$PWD":/workspace \

  macvo:latest bash

Then inside the container, similar to how AIR SLAM needed a specific dataset structure so dose MAC-VO, run this script to create the needed structure:

- python3 ./src/ make_data_4_MAC_VO.py

And then to run the test you need to do this:

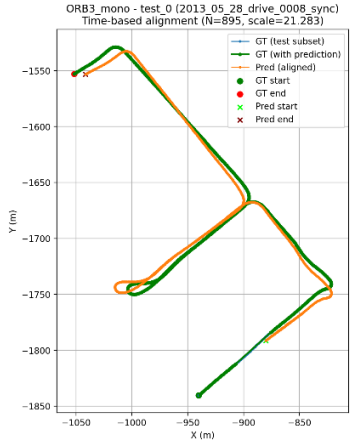
- python3 ./src/run_MAC_VO.py

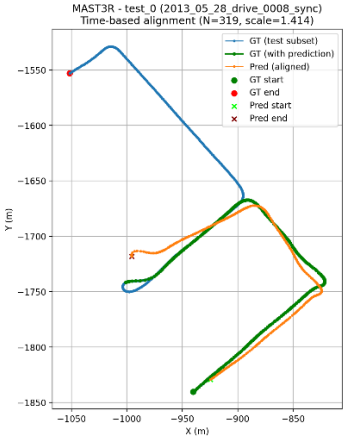
Performance & Observations

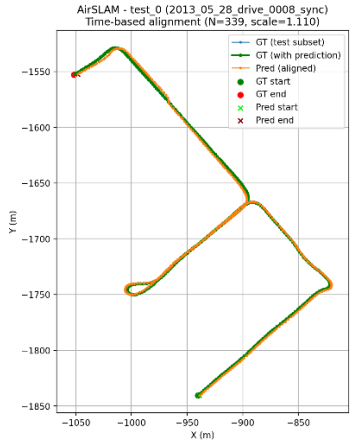
- The algorithm worked, and the results (predicted trajectories) were quite close to ground truth, despite being VO (i.e. without global loop closures).
- GPU usage was high, though the processing speed remained acceptable.
- Because MAC-VO is VO-only (no loop closure / full SLAM backend), and given its GPU demands, I limited testing to the "light" scenario, avoiding full-dataset runs for now.

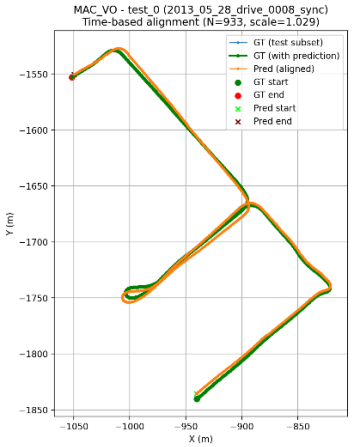
Test$_0$ Summary & Cross-Algorithm Comparison

The tables below consolidate the most important computed metrics: runtime, mapping performance, trajectory errors (ATE/RPE), and system-resource usage. These summaries allow quick identification of differences in robustness, speed, and computational load across the evaluated SLAM algorithms.

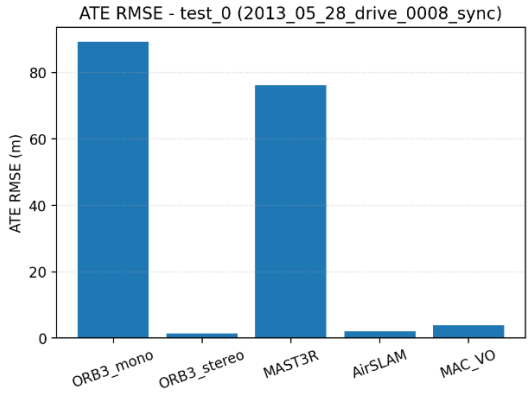


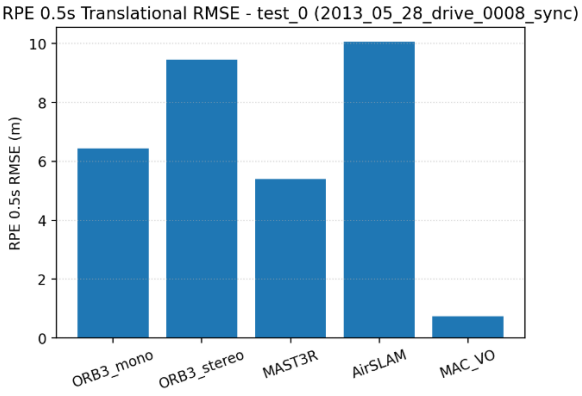




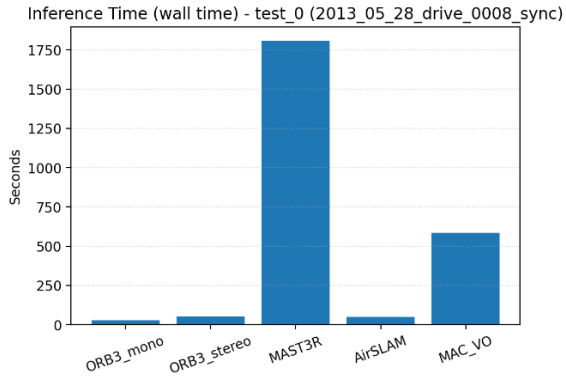


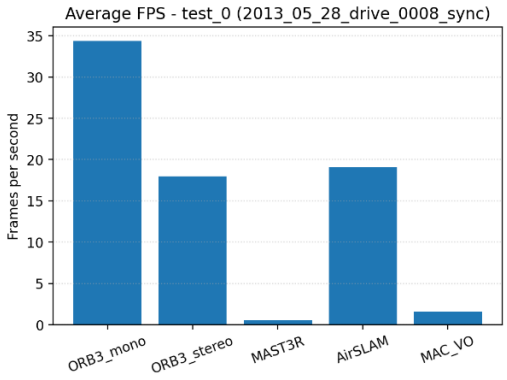
Here you can see the erros in paths prediction:



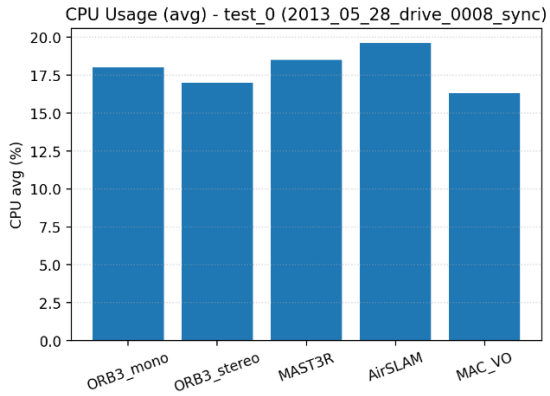


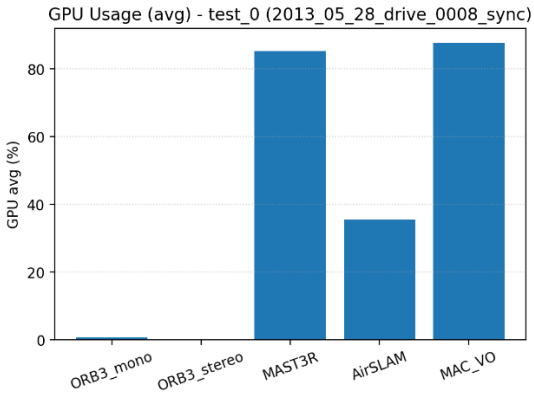
Here you can see the wall time to process the test0 frames:





Here you can see the machine usage:





Condensed Performance Overview:

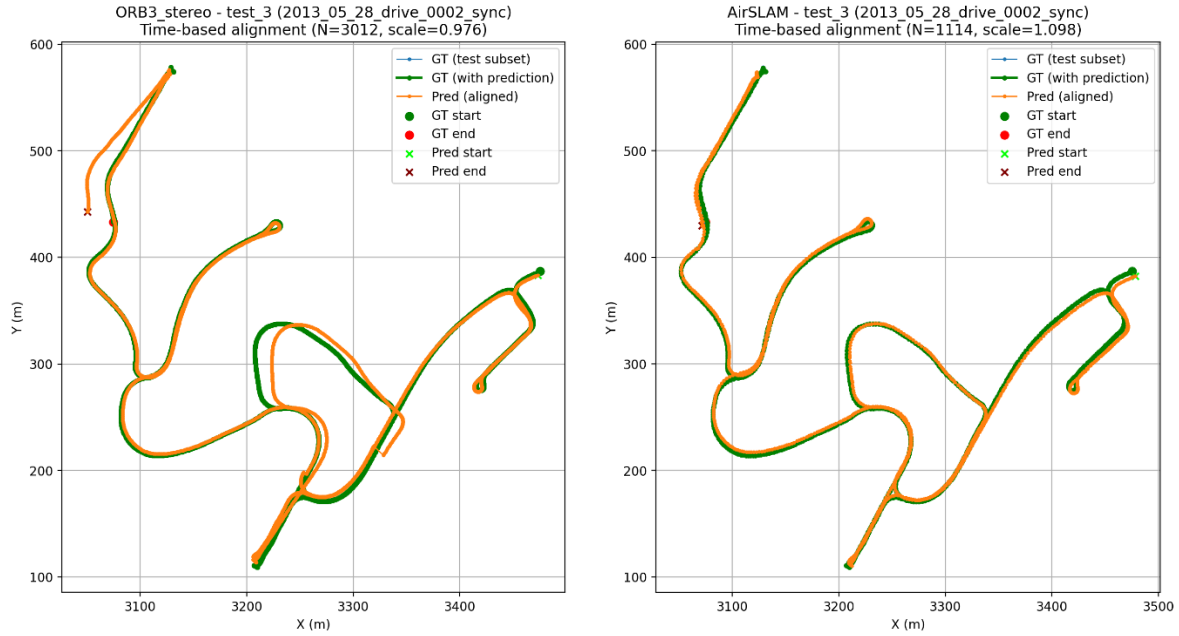
System	ATE RMSE ↓ (m)	FPS ↑	GPU Usage ↑	Notes
ORB Mono	**89.19**	**34.3**	~0%	Fast but extremely inaccurate
ORB Stereo	**1.32**	17.9	~0%	Best accuracy & CPU-friendly
AIR SLAM	**1.97**	19.1	~35%	Good accuracy, more resource use
MACVO	**3.34**	1.6	**87–100%**	Strong RPE, very slow
Master SLAM	**76.20**	0.5	**85–100%**	Slow and inaccurate

Key Takeaways:

- ORB Stereo is the most balanced solution → best accuracy without GPU demand.
- AIR SLAM offers solid accuracy with higher compute cost.
- MACVO has precise relative tracking but impractical speed.
- Master SLAM and ORB Mono perform poorly in accuracy despite opposite runtime speeds.
- For real-time use, ORB Stereo or AIR SLAM are the only viable choices here.

Test₃ Path Comparison & Qualitative Analysis

Test₃ is the **longest sequence** in the KITTI-360 SLAM subset, containing extended trajectories, multiple long loops, and several segments with challenging visual conditions. Because of its length and repeated revisits of the same areas, it is an excellent benchmark for evaluating **long-term consistency**, **loop-closure correctness**, and **overall robustness**.



ORB SLAM tracks well in short and visually rich segments, but its trajectory gradually diverges over long distances. Drift accumulates due to inconsistent loop closures, missed relocalization in low-feature areas, and small lateral errors during turns or blurred motion. It remains locally accurate but lacks global consistency on long loops.

Air SLAM maintains a trajectory that stays closely aligned with ground truth throughout the entire sequence. Reliable loop closures, fast relocalization, and stable tracking during rapid rotations or difficult visual conditions result in minimal drift and only small-scale deviations.