# pySLAM: An Open-Source, Modular, and Extensible Framework for SLAM

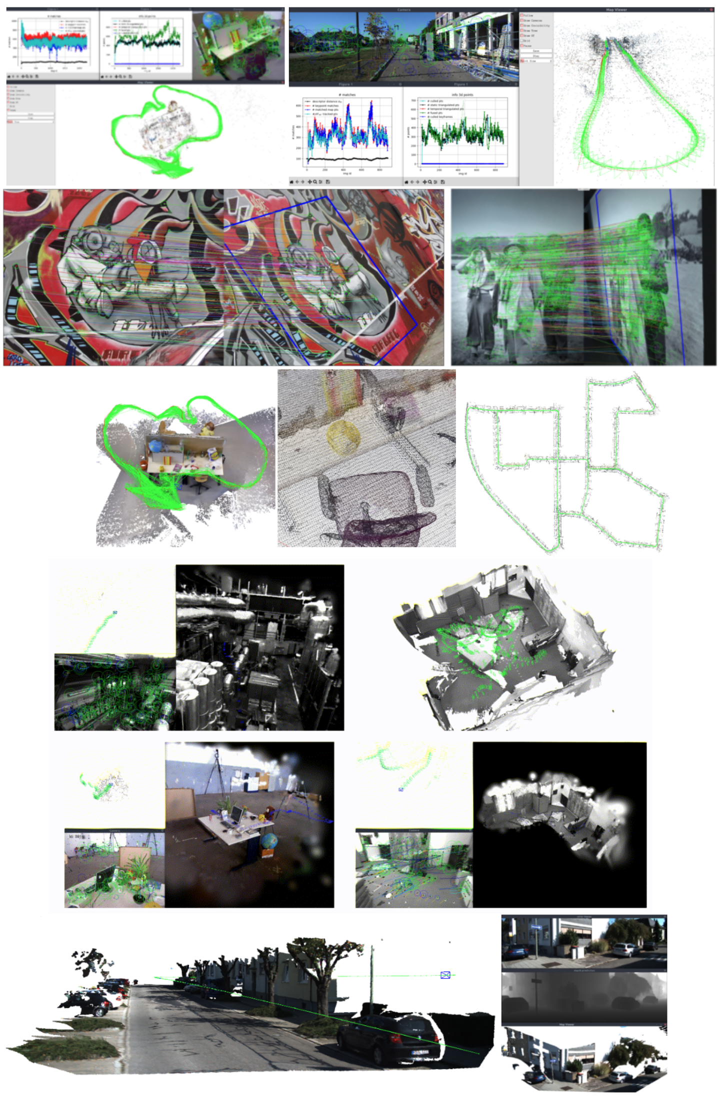
Luigi Freda

April 19, 2025

github.com/luigifreda/pyslam

**Abstract**

pySLAM is an open-source Python framework for Visual SLAM, supporting monocular, stereo, and RGB-D cameras. It provides a flexible interface for integrating both classical and modern local features, making it adaptable to various SLAM tasks. The framework includes different loop closure methods, a volumetric reconstruction pipeline, and support for depth prediction models. Additionally, it offers a suite of tools for visual odometry and SLAM applications. Designed for both beginners and experienced researchers, pySLAM encourages community contributions, fostering collaborative development in the field of Visual SLAM. This document presents the pySLAM framework, its main features, and usage.



# Contents

# 1 Introduction

The objective of this document is to present the pySLAM framework, its main features, and usage[1]. **pySLAM** is a python implementation of a *Visual SLAM* pipeline that supports **monocular**, **stereo** and **RGBD** cameras. It provides the following **features** in a single python environment:

- A wide range of classical and modern **local features** with a convenient interface for their integration.
- Various loop closing methods, including **descriptor aggregators** such as visual Bag of Words (BoW, iBow), Vector of Locally Aggregated Descriptors (VLAD), and modern **global descriptors** (image-wise descriptors).
- A **volumetric reconstruction pipeline** that processes available depth and color images with volumetric integration and provides an output dense reconstruction. This can use **TSDF** with voxel hashing or incremental **Gaussian Splatting**.
- Integration of **depth prediction models** within the SLAM pipeline. These include DepthPro, DepthAnythingV2, RAFT-Stereo, CREStereo, MASt3R, MVDUSt3R, etc.
- Additional tools for VO (Visual Odometry) and SLAM, with built-in support for both **g2o** and **GTSAM**, along with custom Python bindings for features not included in the original libraries
- Built-in support for over **10 dataset types**.

pySLAM serves as flexible baseline framework to experiment with VO/SLAM techniques, *local features*, *descriptor aggregators*, *global descriptors*, *volumetric integration* and *depth prediction*. It allows to explore, prototype and develop VO/SLAM pipelines. pySLAM is a research framework and a work in progress. It is not optimized for real-time performances.

**Enjoy it!**

# 2 Overview

## 2.1 Main Scritps

A convenient entry-point are the following **main scripts**:

- `main_vo.py` combines the simplest VO ingredients without performing any image point triangulation or windowed bundle adjustment. At each step $k$, `main_vo.py` estimates the current camera pose $C_k$ with respect to the previous one $C_{k-1}$. The inter-frame pose estimation returns $[R_{k-1,k}, t_{k-1,k}]$ with $\|t_{k-1,k}\| = 1$. With this very basic approach, you need to use a ground truth in order to recover a correct inter-frame scale $s$ and estimate a valid trajectory by composing $C_k = C_{k-1}[R_{k-1,k}, st_{k-1,k}]$. This script is a first start to understand the basics of inter-frame feature tracking and camera pose estimation.
- `main_slam.py` adds feature tracking along multiple frames, point triangulation, keyframe management, bundle adjustment, loop closing, dense mapping and depth inference in order to estimate the camera trajectory and build both a sparse and dense map. It's a full SLAM pipeline and includes all the basic and advanced blocks which are necessary to develop a real visual SLAM pipeline.
- `main_feature_matching.py` shows how to use the basic feature tracker capabilities (*feature detector* + *feature descriptor* + *feature matcher*) and allows to test the different available local features.
- `main_depth_prediction.py` shows how to use the available depth inference models to get depth estimations from input color images.
- `main_map_viewer.py` reloads a saved map and visualizes it. Further details on how to save a map here.
- `main_map_dense_reconstruction.py` reloads a saved map and uses a configured volumetric integrator to obtain a dense reconstruction (see here).
- `main_slam_evaluation.py` enables automated SLAM evaluation by executing `main_slam.py` across a collection of datasets and configuration presets (see here).

[1]You may find an updated version of this document at:
github.com/luigifreda/pyslam/blob/master/docs/tex/document.pdf

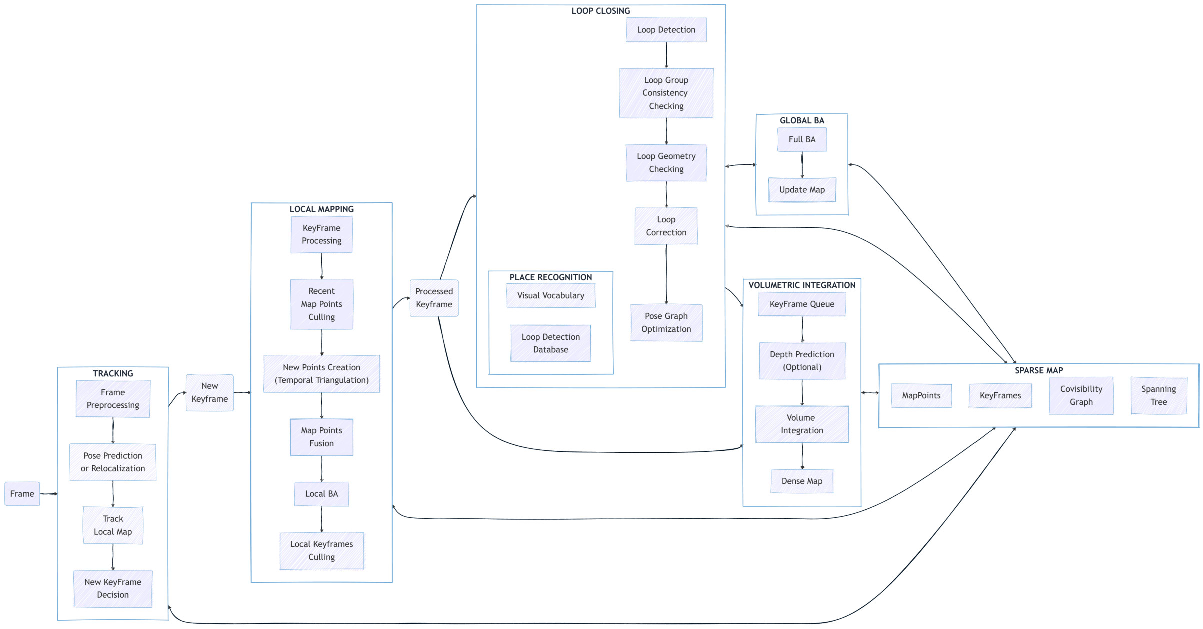



Figure 1: SLAM workflow.

## 2.2 System overview

This section presents some diagram sketches that provide an overview of the main workflow, system components, and class relationships/dependencies. To make the diagrams more readable, some minor components and arrows have been omitted.

### SLAM Workflow and Components

Fig. 1 illustrates the SLAM workflow, which is composed of five main parallel processing modules:

- *Tracking*: estimates the camera pose for each incoming frame by extracting and matching local features to the local map, followed by minimizing the reprojection error through motion-only Bundle Adjustment (BA). It includes components such as pose prediction (or relocalization), feature tracking, local map tracking, and keyframe decision-making.
- *Local Mapping*: updates and refines the local map by processing new keyframes. This involves culling redundant map points, creating new points via temporal triangulation, fusing nearby map points, performing Local BA, and pruning redundant local keyframes.
- *Loop Closing*: detects and validates loop closures to correct drift accumulated over time. Upon loop detection, it performs loop group consistency checks and geometric verification, applies corrections, and then launches Pose Graph Optimization (PGO) followed by a full Global Bundle Adjustment (GBA). Loop detection itself is delegated to a parallel process, the *Loop Detector*, which operates independently for better responsiveness and concurrency.
- *Global Bundle Adjustment*: triggered by the Loop Closing module after PGO, this step globally optimizes the trajectory and the sparse structure of the map to ensure consistency across the entire sequence.
- *Volumetric Integration*: uses the keyframes, with their estimated poses and back-projected point clouds, to reconstruct a dense 3D map of the environment. This module optionally integrates predicted depth maps and maintains a volumetric representation such as a TSDF [13] or incremental Gaussian Splatting-based volume [34, 19].

The first four modules follow the established PTAM [20] and ORB-SLAM [39] paradigm. Here, the *Tracking* module serves as the front-end, while the remaining modules operate as part of the back-end.

In parallel, the system constructs two types of maps:

- a *sparse map* $\mathcal{M}_s = (\mathcal{K}, \mathcal{P})$, composed of a set of keyframes $\mathcal{K}$ and 3D points $\mathcal{P}$ derived from matched features;
- a *volumetric map* (or dense map) $\mathcal{M}_v$, constructed by the Volumetric Integration module, which fuses back-projected point clouds from the keyframes $\mathcal{K}$ into a dense 3D model.

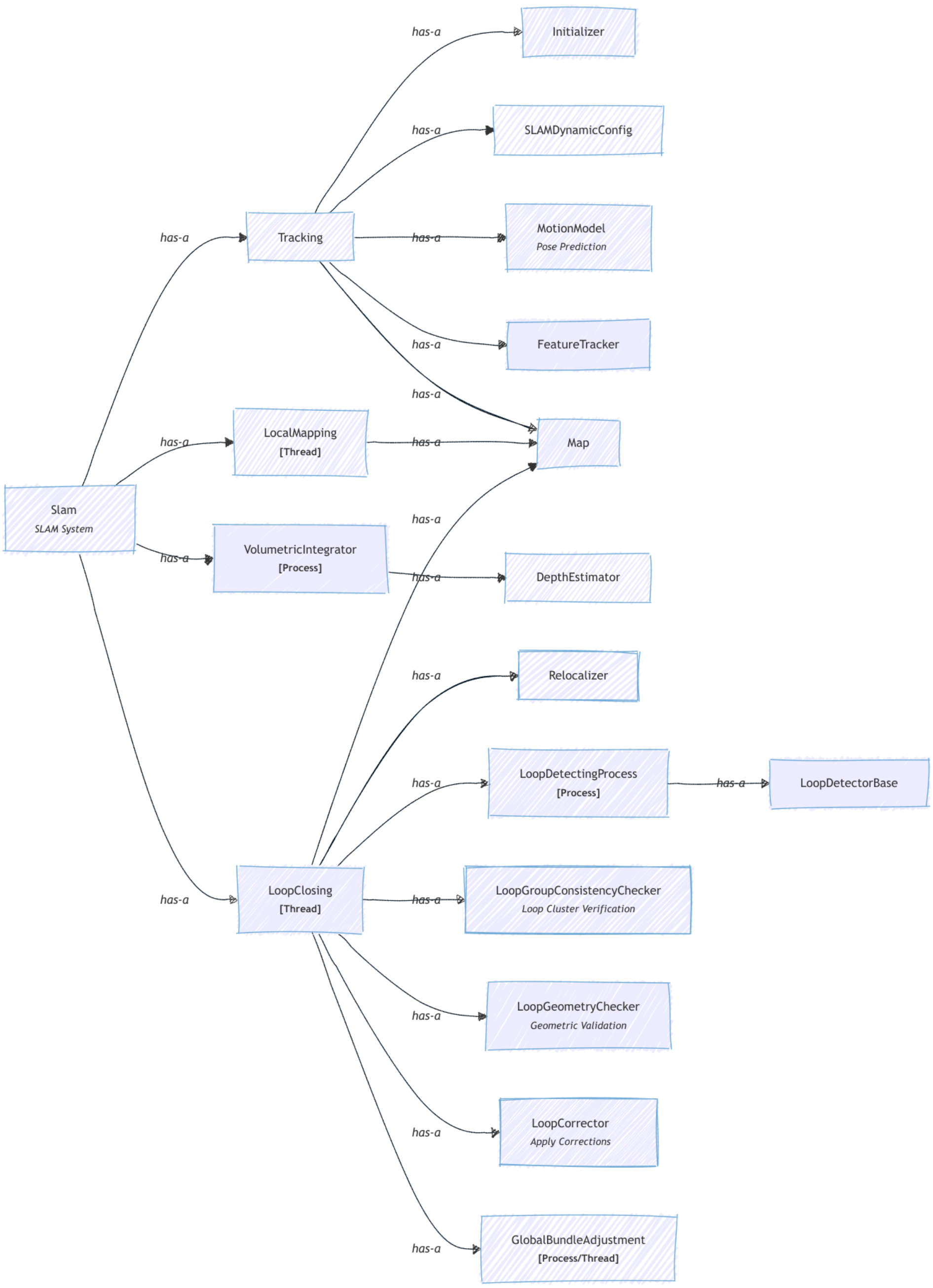



Figure 2: SLAM components.

To ensure consistency between the sparse and volumetric representations, the volumetric map is updated or re-integrated whenever global pose adjustments occur (e.g., after loop closures).

Fig. 2 details the internal components and interactions of the above modules. In certain cases, **processes** are employed instead of **threads**. This is due to Python's Global Interpreter Lock (GIL), which prevents concurrent execution of multiple threads in a single process. The use of multiprocessing circumvents this limitation, enabling true parallelism at the cost of some inter-process communication overhead (e.g., via pickling). For an insightful discussion, see this related post.

## 2.3 Main System Components

### Feature Tracker

The *Feature Tracker* consists of the following key sub-components:

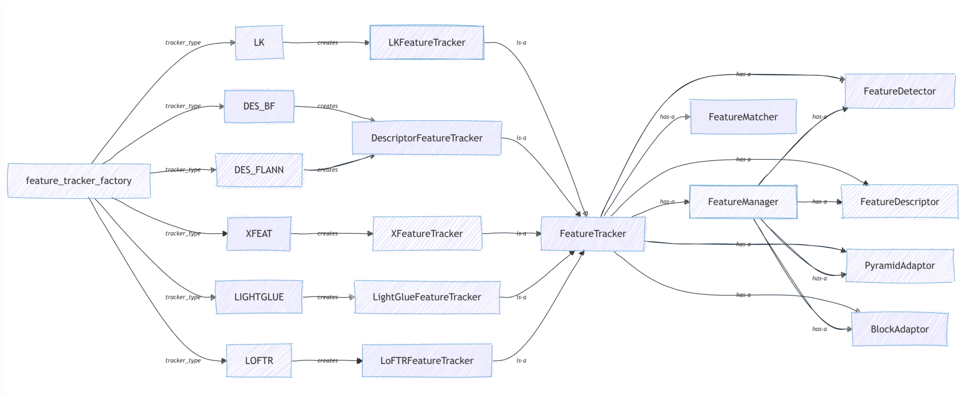



Figure 3: Feature tracker.

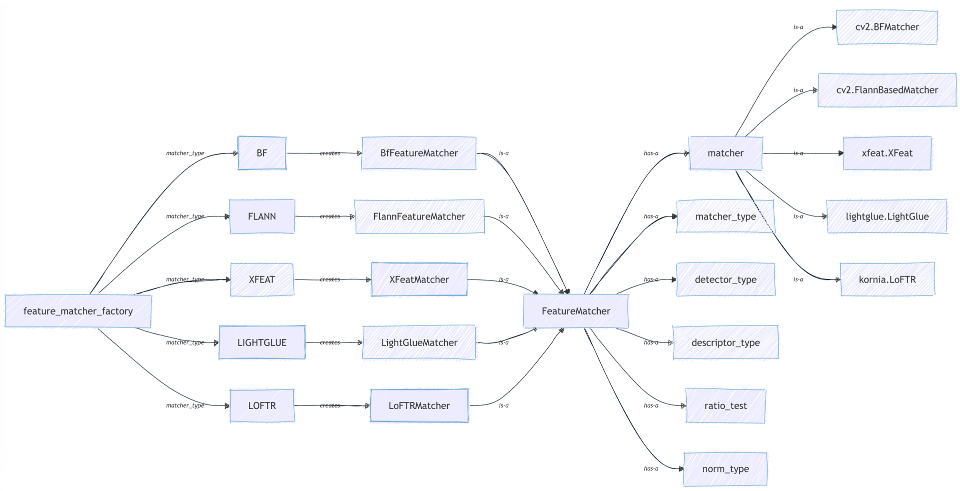



Figure 4: Feature matcher.

- *Feature Extractor*: identifies salient and repeatable keypoints in the image, such as corners or blobs, which are likely to be robust under viewpoint and illumination changes.
- *Feature Detector*: computes a distinctive descriptor for each detected keypoint, encoding its local appearance to enable robust matching across frames. Examples include ORB [50], SIFT [28], or SuperPoint [12] descriptors.
- *Feature Matcher*: establishes correspondences between features in successive frames (or stereo pairs) by comparing their descriptors. Matching can be performed using brute-force, k-NN with ratio test, or learned matching strategies. Refere to Sect. 2.3 for futher details.

Sect. 4.1 reports the list of supported local feature extractors and detectors.

The diagram in Fig. 3 presents the architecture of the *Feature Tracker* system. It is structured around a `feature_tracker_factory`, which instantiates specific tracker types such as `LK`, `DES_BF`, `DES_FLANN`, `XFEAT`, `LIGHTGLUE`, and `LOFTR`. Each tracker type creates a corresponding implementation (e.g., `LKFeatureTracker`, `DescriptorFeatureTracker`, etc.), all of which inherit from a common `FeatureTracker` interface.

The `FeatureTracker` class is composed of several key sub-components, including a `FeatureManager`, `FeatureDetector`, `FeatureDescriptor`, `PyramidAdaptor`, `BlockAdaptor`, and `FeatureMatcher`. The `FeatureManager` itself also encapsulates instances of the detector, descriptor, and adaptors, highlighting the modular and reusable design of the tracking pipeline.

### Feature Matcher

The diagram in Fig. 4 illustrates the architecture of the *Feature Matcher* module. At its core is the `feature_matcher_factory`, which instantiates matchers based on a specified `matcher_type`, such as `BF`, `FLANN`, `XFEAT`, `LIGHTGLUE`, and `LOFTR`. Each of these creates a corresponding matcher implementation (e.g., `BfFeatureMatcher`, `FlannFeatureMatcher`, etc.), all inheriting from a common `FeatureMatcher` interface.

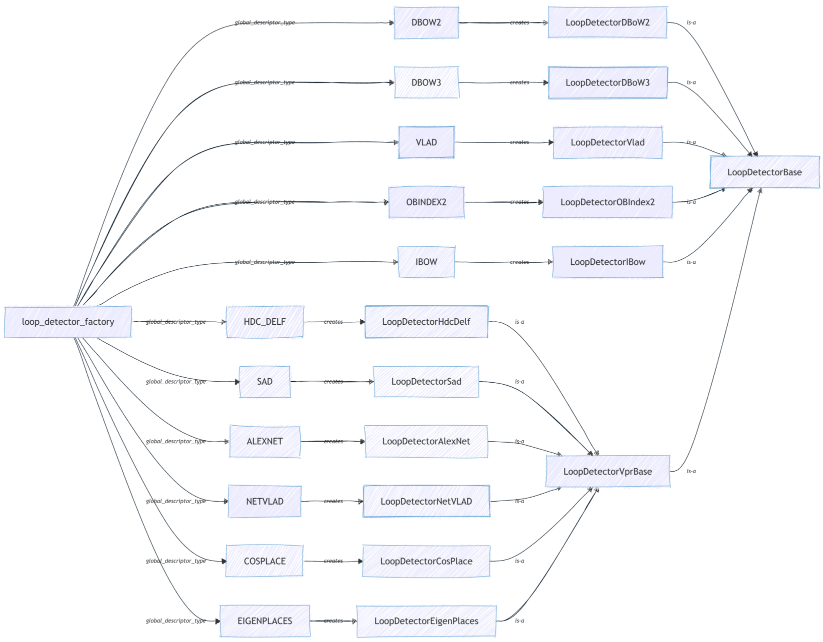



Figure 5: Loop detector.

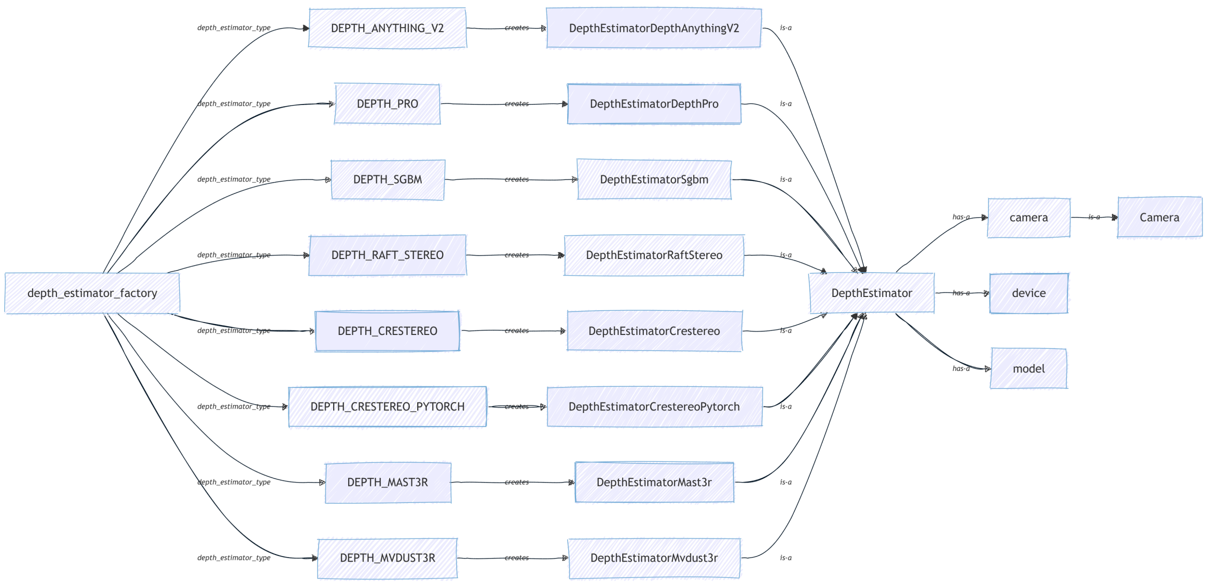



Figure 6: Depth estimator.

The `FeatureMatcher` class encapsulates several configuration parameters and components, including the matcher engine (`cv2.BFMatcher`, `FlannBasedMatcher`, `xfeat.XFeat`, etc.), as well as the `matcher_type`, `detector_type`, `descriptor_type`, `norm_type`, and `ratio_test` fields. This modular structure supports extensibility and facilitates switching between traditional and learning-based feature matching backends.

The Section 4.2 reports a list of supported feature matchers.

### Loop Detector

The diagram in Fig. 5 shows the architecture of the *Loop Detector* component. A central `loop_detector_factory` instantiates loop detectors based on the selected `global_descriptor_type`, which may include traditional descriptors (e.g., `DBOW2`, `VLAD`, `IBOW`) or deep learning-based embeddings (e.g., `NetVLAD`, `CosPlace`, `EigenPlaces`).

Each descriptor type creates a corresponding loop detector implementation (e.g., `LoopDetectorDBoW2`, `LoopDetectorNetVLAD`), all of which inherit from a base class hierarchy. Traditional methods inherit directly from `LoopDetectorBase`, while deep learning-based approaches inherit from `LoopDetectorVprBase`, which itself

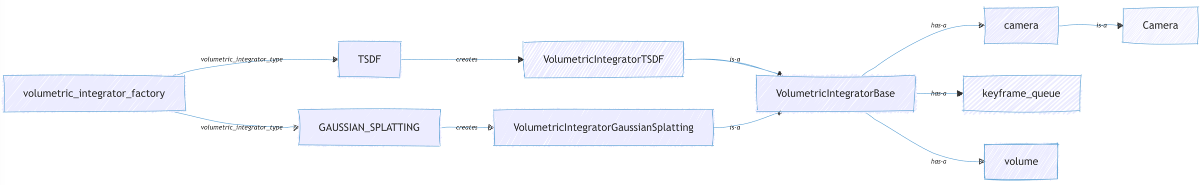



Figure 7: Volumetric integrator.

extends `LoopDetectorBase`. This design supports modular integration of diverse place recognition techniques within a unified loop closure framework.

The Section 4.3 reports a list of supported loop closure methods with the adopted global descriptors and local descriptor aggregation methods.

### Depth Estimator

The diagram in Fig. 6 illustrates the architecture of the *Depth Estimator* module. A central `depth_estimator_factory` creates instances of various depth estimation backends based on the selected `depth_estimator_type`, including both traditional and learning-based methods such as `DEPTH_SGBM`, `DEPTH_RAFT_STEREO`, `DEPTH_ANYTHING_V2`, `DEPTH_MAST3R`, and `DEPTH_MVDUST3R`.

Each estimator type instantiates a corresponding implementation (e.g., `DepthEstimatorSgbm`, `DepthEstimatorCrestereo`, etc.), all inheriting from a common `DepthEstimator` interface. This base class encapsulates shared dependencies such as the `camera`, `device`, and `model` components, allowing for modular integration of heterogeneous depth estimation techniques across stereo, monocular, and multi-view pipelines.

The Section 4.4 reports a list of supported depth estimation/prediction models.

### Volumetric Integrator

The diagram in Fig. 7 illustrates the structure of the *Volumetric Integrator* module. At its core, the `volumetric_integrator_factory` generates specific volumetric integrator instances based on the selected `volumetric_integrator_type`, such as `TSDF` and `GAUSSIAN_SPLATTING`.

Each type instantiates a dedicated implementation (e.g., `VolumetricIntegratorTSDF`, `VolumetricIntegrator-GaussianSplatting`), which inherits from a common `VolumetricIntegratorBase`. This base class encapsulates key components including the `camera`, a `keyframe_queue`, and the `volume`, enabling flexible integration of various 3D reconstruction methods within a unified pipeline.

The Section 4.5 reports a list of supported volume integration methods.

# 3 Usage

Open a new terminal and start experimenting with the scripts. In each new terminal you are supposed to start with this command:

```
$ . pyenv-activate.sh   #  Activate pyslam python virtual environment. This is only needed once in a new terminal.
```

The file config.yaml can be used as a unique entry-point to configure the system and its global configuration parameters contained in config_parameters.py. Further information on how to configure pySLAM are provided here.

### Visual odometry

The basic **Visual Odometry** (VO) can be run with the following commands:

```
$ . pyenv-activate.sh   #  Activate pyslam python virtual environment. This is only needed once in a new terminal.
$ ./main_vo.py
```

By default, this processes a KITTI video (available in the folder `data/videos`) by using its corresponding camera calibration file (available in the folder `settings`), and its groundtruth (available in the same `data/videos` folder). If matplotlib windows are used, you can stop `main_vo.py` by focusing/clicking on one of them and pressing the key 'Q'. As explained above, this very *basic* script `main_vo.py` **strictly requires a ground truth**. Now, with RGBD datasets, you can also test the **RGBD odometry** with the classes `VisualOdometryRgbd` or `VisualOdometryRgbdTensor` (ground truth is not required here).

### Full SLAM

Similarly, you can test the **full SLAM** by running `main_slam.py`:

```
$ . pyenv-activate.sh   #  Activate pyslam python virtual environment. This is only needed once in a new terminal.
$ ./main_slam.py
```

This will process the same default KITTI video (available in the folder `data/videos`) by using its corresponding camera calibration file (available in the folder `settings`). You can stop it by focusing/clicking on one of the opened windows and pressing the key 'Q' or closing the 3D pangolin GUI.

#### Selecting a dataset and different configuration parameters

The file config.yaml can be used as a unique entry-point to configure the system, the target dataset and its global configuration parameters set in config_parameters.py. To process a different **dataset** with both VO and SLAM scripts, you need to update the file config.yaml:

- Select your dataset `type` in the section `DATASET` (further details in the section *Datasets* below for further details). This identifies a corresponding dataset section (e.g. `KITTI_DATASET`, `TUM_DATASET`, etc).
- Select the `sensor_type` (`mono`, `stereo`, `rgbd`) in the chosen dataset section.
- Select the camera `settings` file in the dataset section (further details in the section *Camera Settings* below).
- Set the `groudtruth_file` accordingly. Further details in the section *Datasets* below (see also the files `io/ground_truth.py`, `io/convert_groundtruth_to_simple.py`).

You can use the section `GLOBAL_PARAMETERS` of the file config.yaml to override the global configuration parameters set in config_parameters.py. This is particularly useful when running a SLAM evaluation.

## 3.1 Feature tracking

If you just want to test the basic feature tracking capabilities (*feature detector* + *feature descriptor* + *feature matcher*) and get a taste of the different available local features, run

```
$ . pyenv-activate.sh   #  Activate pyslam python virtual environment. This is only needed once in a new terminal.
$./main_feature_matching.py
```

In any of the above scripts, you can choose any detector/descriptor among *ORB*, *SIFT*, *SURF*, *BRISK*, *AKAZE*, *SuperPoint*, etc. (see the section *Supported Local Features* below for further information).

Some basic examples are available in the subfolder `test/cv`. In particular, as for feature detection/description, you may want to take a look at test/cv/test_feature_manager.py too.

## 3.2 Loop closing

Many loop closing methods are available, combining different aggregation methods and global descriptors.

While running full SLAM, loop closing is enabled by default and can be disabled by setting `kUseLoopClosing=False` in `config_parameters.py`. Different configuration options `LoopDetectorConfigs` can be found in loop_closing/loop_detector_configs.py: Code comments provide additional useful details.

One can start experimenting with loop closing methods by using the examples in `test/loopclosing`. The example test/loopclosing/test_loop_detector.py is the recommended entry point.

#### Vocabulary management

`DBoW2`, `DBoW3`, and `VLAD` require **pre-trained vocabularies**. ORB-based vocabularies are automatically downloaded in the `data` folder (see loop_closing/loop_detector_configs.py).

To create a new vocabulary, follow these steps:

1. **Generate an array of descriptors**: Use the script `test/loopclosing/test_gen_des_array_from_imgs.py` to generate the array of descriptors that will be used to train the new vocabulary. Select your desired descriptor type via the tracker configuration.
2. **DBOW vocabulary generation**: Train your target DBOW vocabulary by using the script `test/loopclosing/test_gen_dbow_voc_from_des_array.py`.
3. **VLAD vocabulary generation**: Train your target VLAD "vocabulary" by using the script `test/loopclosing/test_gen_vlad_voc_from_des_array.py`.

Once you have trained the vocabulary, you can add it in loop_closing/loop_detector_vocabulary.py and correspondingly create a new loop detector configuration in loop_closing/loop_detector_configs.py that uses it.

#### Vocabulary-free loop closing

Most methods do not require pre-trained vocabularies. Specifically:

- `iBoW` and `OBindex2`: These methods incrementally build bags of binary words and, if needed, convert (front-end) non-binary descriptors into binary ones.
- Others: Methods like `HDC_DELF`, `SAD`, `AlexNet`, `NetVLAD`, `CosPlace`, and `EigenPlaces` directly extract their specific **global descriptors** and process them using dedicated aggregators, independently from the used front-end descriptors.

As mentioned above, only `DBoW2`, `DBoW3`, and `VLAD` require pre-trained vocabularies.

#### Double-check your loop detection configuration and verify vocabulary compability

#### Loop detection method based on a pre-trained vocabulary

When selecting a **loop detection method based on a pre-trained vocabulary** (such as `DBoW2`, `DBoW3`, and `VLAD`), ensure the following:

1. The back-end and the front-end are using the same descriptor type (this is also automatically checked for consistency) or their descriptor managers are independent (see further details in the configuration options `LoopDetectorConfigs` available in loop_closing/loop_detector_configs.py).
2. A corresponding pre-trained vocubulary is available. For more details, refer to the vocabulary management section.

#### Missing vocabulary for the selected front-end descriptor type

If you lack a compatible vocabulary for the selected front-end descriptor type, you can follow one of these options:
1. Create and load the vocabulary (refer to the vocabulary management section).
2. Choose an `*_INDEPENDENT` loop detector method, which works with an independent local_feature_manager.
3. Select a vocabular-free loop closing method.

See the file loop_closing/loop_detector_configs.py for further details.

## 3.3 Volumetric reconstruction

#### Dense reconstruction while running SLAM

The SLAM back-end hosts a volumetric reconstruction pipeline. This is disabled by default. You can enable it by setting `kUseVolumetricIntegration=True` and selecting your preferred method `kVolumetricIntegrationType` in `config_parameters.py`. At present, two methods are available: `TSDF` and `GAUSSIAN_SPLATTING` (see dense/volumetric_integrator_factory.py). Note that you need CUDA in order to run `GAUSSIAN_SPLATTING` method.

At present, the volumetric reconstruction pipeline works with:

- RGBD datasets
- When a depth estimator is used
  - in the back-end with STEREO datasets (you can't use depth prediction in the back-end with MONOCULAR datasets, further details here)
  - in the front-end (to emulate an RGBD sensor) and a depth prediction/estimation gets available for each processed keyframe.

If you want a mesh as output then set `kVolumetricIntegrationExtractMesh=True` in `config_parameters.py`.

#### Reload a saved sparse map and perform dense reconstruction

Use the script `main_map_dense_reconstruction.py` to reload a saved sparse map and to perform dense reconstruction by using its posed keyframes as input. You can select your preferred dense reconstruction method directly in the script.

- To check what the volumetric integrator is doing, run in another shell `tail -f logs/volumetric_integrator.log` (from repository root folder).
- To save the obtained dense and sparse maps, press the `Save` button on the GUI.

#### Reload and check your dense reconstruction

You can check the output pointcloud/mesh by using CloudCompare.

In the case of a saved Gaussian splatting model, you can visualize it by:

1. Using the superslat editor (drag and drop the saved Gaussian splatting `.ply` pointcloud in the editor interface).
2. Getting into the folder `test/gaussian_splatting` and running:

   ```
   $ python test_gsm.py --load <gs_checkpoint_path>
   ```

   The `<gs_checkpoint_path>` is expected to have the following structure:

   ```
   +-- gs_checkpoint_path
   |   +-- pointcloud         # folder containing different subfolders, each one with a saved .ply
   |                          # encoding the gaussian splatting model at a specific iteration/checkpoint
   |   +-- last_camera.json
   |   +-- config.yml
   ```

#### Controlling the spatial distribution of keyframe FOV centers

If you are targeting volumetric reconstruction while running SLAM, you can enable a **keyframe generation policy** designed to manage the spatial distribution of keyframe field-of-view (FOV) centers. The *FOV center of a camera* is defined as the backprojection of its image center, calculated using the median depth of the frame. With this policy, a new keyframe is generated only if its FOV center is farther than a predefined distance from the nearest existing keyframe's FOV center. You can enable this policy by setting the following parameters in the yaml setting:

```
KeyFrame.useFovCentersBasedGeneration: 1  # compute 3D fov centers of camera frames by using median depth
                                          # use their distances to control keyframe generation
KeyFrame.maxFovCentersDistance: 0.2       # max distance between fov centers in order to generate a keyframe
```

#### Depth prediction

The available depth prediction models can be utilized both in the SLAM back-end and front-end.

- **Back-end**: Depth prediction can be enabled in the volumetric reconstruction pipeline by setting the parameter `kVolumetricIntegrationUseDepthEstimator=True` and selecting your preferred `kVolumetricIntegrationDepthEstimatorType` in `config_parameters.py`.
- **Front-end**: Depth prediction can be enabled in the front-end by setting the parameter `kUseDepthEstimatorInFrontEnd` in `config_parameters.py`. This feature estimates depth images from input color images to emulate a RGBD camera. Please, note this functionality is still *experimental* at present time [WIP].

**Notes**:

- In the case of a **monocular SLAM**, do NOT use depth prediction in the back-end volumetric integration: The SLAM (fake) scale will conflict with the absolute metric scale of depth predictions. With monocular datasets, you can enable depth prediction to run in the front-end (to emulated an RGBD sensor).
- The depth inference may be very slow (for instance, with DepthPro it takes ~1s per image on my machine). Therefore, the resulting volumetric reconstruction pipeline may be very slow.

Refer to the file `depth_estimation/depth_estimator_factory.py` for further details. Both stereo and monocular prediction approaches are supported. You can test depth prediction/estimation by using the script `main_depth_prediction.py`.

### 3.4 Saving and reloading

#### Save the a map

When you run the script `main_slam.py` (`main_map_dense_reconstruction.py`):

- You can save the current map state by pressing the button `Save` on the GUI. This saves the current map along with front-end, and backend configurations into the default folder `results/slam_state` (`results/slam_state_dense_reconstruction`).
- To change the default saving path, open `config.yaml` and update target `folder_path` in the section:

  ```
  SYSTEM_STATE:
    folder_path: results/slam_state   # default folder path (relative to repository root) where the system state is saved or reloaded
  ```

### Reload a saved map and relocalize in it

- A saved map can be loaded and visualized in the GUI by running:

```
$ . pyenv-activate.sh   #  Activate pyslam python virtual environment. This is only needed once in a new terminal.
$ ./main_map_viewer.py  #  Use the --path options to change the input path
```

- To enable map reloading and relocalization when running `main_slam.py`, open `config.yaml` and set

```
SYSTEM_STATE:
  load_state: True                # flag to enable SLAM state reloading (map state + loop closing state)
  folder_path: results/slam_state   # default folder path (relative to repository root) where the system state is saved or reloaded
```

Note that pressing the `Save` button saves the current map, front-end, and backend configurations. Reloading a saved map overwrites the current system configurations to ensure descriptor compatibility.

### Trajectory saving

Estimated trajectories can be saved in three different formats: *TUM* (The Open Mapping format), *KITTI* (KITTI Odometry format), and *EuRoC* (EuRoC MAV format). pySLAM saves two **types** of trajectory estimates:

- **Online**: In *online* trajectories, each pose estimate depends only on past poses. A pose estimate is saved at the end of each front-end iteration on current frame.
- **Final**: In *final* trajectories, each pose estimate depends on both past and future poses. A pose estimate is refined multiple times by LBA windows that cover it and by GBA during loop closures.

To enable trajectory saving, open `config.yaml` and search for the `SAVE_TRAJECTORY`: set `save_trajectory: True`, select your `format_type` (`tum`, `kitti`, `euroc`), and the output filename. For instance for a `tum` format output:

```
SAVE_TRAJECTORY:
  save_trajectory: True
  format_type: kitti             # supported formats: `tum`, `kitti`, `euroc`
  output_folder: results/metrics # relative to pyslam root folder
  basename: trajectory           # basename of the trajectory saving output
```

### Optimization engines

Currently, pySLAM supports both `g2o` and `gtsam` for graph optimization, with `g2o` set as the default engine. While `gtsam` is fully supported, it remains experimental and a work in progress. You can enable `gtsam` by setting to `True` the following parameters in `config_parameters.py`:

```
      # Optimization engine
      kOptimizationFrontEndUseGtsam = True
      kOptimizationBundleAdjustUseGtsam = True
      kOptimizationLoopClosingUseGtsam = True
```

Additionally, the `gtsam_factors` package provides custom Python bindings for features not included in the original gtsam framework. See here for further details.

## 3.5 SLAM GUI

Some quick information about the non-trivial GUI buttons of `main_slam.py`:

- `Step`: Enter the *Step by step mode*. Press the button `Step` a first time to pause. Then, press it again to make the pipeline process a single new frame.
- `Save`: Save the map into the file `map.json`. You can visualize it back by using the script `main_map_viewer.py` (as explained above).
- `Reset`: Reset SLAM system.
- `Draw Ground Truth`: If a ground truth dataset (e.g., KITTI, TUM, EUROC, or REPLICA) is loaded, you can visualize it by pressing this button. The ground truth trajectory will be displayed in 3D and will be progressively aligned with the estimated trajectory, updating approximately every 10-30 frames. As more frames are processed, the alignment between the ground truth and estimated trajectory becomes more accurate. After about 20 frames, if the button is pressed, a window will appear showing the Cartesian alignment errors along the main axes (i.e., $e_x, e_y, e_z$ and the history of the total $RMSE$ between the ground truth and the aligned estimated trajectories.

## 3.6 Monitor the logs for tracking, local mapping, and loop closing simultaneously

The logs generated by the modules `local_mapping.py`, `loop_closing.py`, `loop_detecting_process.py`, `global_bundle_adjustments.py`, and `volumetric integrator_<X>.py` are collected in the files `local_mapping.log`, `loop_closing.log`, `loop_detecting.log`, `gba.log`, and `volumetric_integrator.log`, which are all stored in the folder `logs`.

For debugging, you can monitor one of the them in parallel by running the following command in a separate shell:

```
$ tail -f logs/<log file name>
```

Otherwise, to check all parallel logs with tmux, run:

```
$ ./scripts/launch_tmux_logs.sh
```

To launch slam and check all logs in a single tmux, run:

```
$ ./scripts/launch_tmux_slam.sh
```

Press `CTRL+A` and then `CTRL+Q` to exit from `tmux` environment.

## 3.7 Evaluating SLAM

#### Run a SLAM evaluation

The `main_slam_evaluation.py` script enables automated SLAM evaluation by executing `main_slam.py` across a collection of datasets and configuration presets. The input evaluation configuration file (e.g., `evaluation/configs/evaluation.json`) specifies the datasets and presets to be used. For each evaluation run, results are stored in a dedicated subfolder within the `results` directory, containing all the computed metrics. These metrics are then processed and compared. The final output is a report, available in both PDF, LaTeX, and HTML formats, that includes comparison tables summarizing the *Absolute Trajectory Error* (ATE), the maximum deviation from the ground truth trajectory and other metrics.

You can find some obtained evaluation results on this local page or at this absolute link.

#### pySLAM performances and comparative evaluations

For a comparative evaluation of the "**online**" trajectory estimated by pySLAM versus the "**final**" trajectory estimated by ORB-SLAM3, check out this nice notebook. For more details about "*online*" and "*final*" trajectories, refer to this section.

**Note**: Unlike ORB-SLAM3, which only saves the final pose estimates (recorded after the entire dataset has been processed), pySLAM saves both online and final pose estimates. For details on how to save trajectories in pySLAM, refer to this section. When you click the `Draw Ground Truth` button in the GUI (see here), you can visualize the *Absolute Trajectory Error* (ATE or *RMSE*) history and evaluate both online and final errors up to the current time.

# 4 Supported components and models

## 4.1 Supported local features

Fig. 8 shows a timeline with some of the most famous local features for image matching, place recognition and SLAM. At present time, pySLAM supports the following feature **detectors**:

- FAST [49]
- Good features to track [52]
- ORB [50]
- ORB2 (improvements of ORB-SLAM2 to ORB detector)
- SIFT [28]
- SURF [8]
- KAZE [1]
- AKAZE [2]

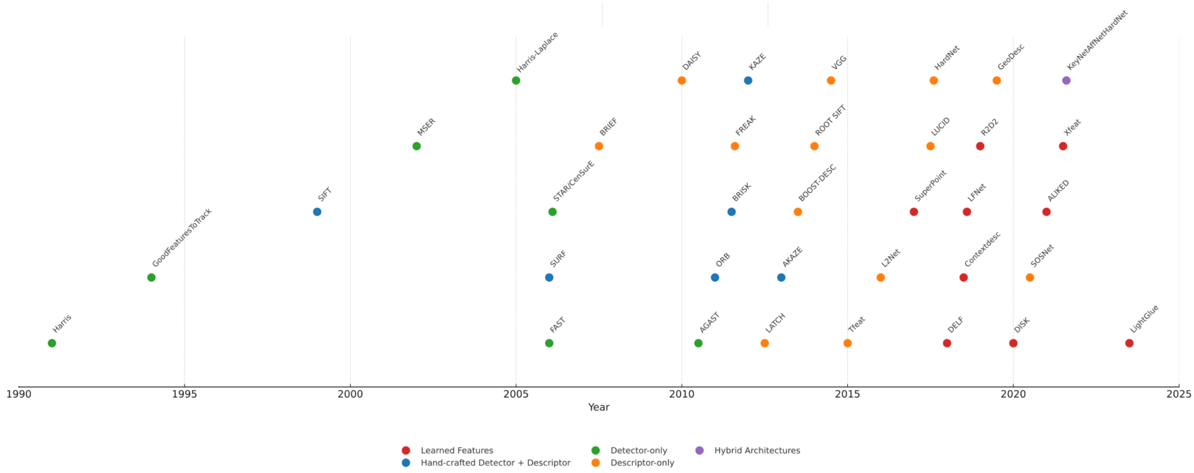



Figure 8: Timeline of some of the most famous local features for image matching, place recognition and SLAM.

- BRISK [22]
- AGAST
- MSER [33]
- StarDector/CenSurE
- Harris-Laplace
- SuperPoint
- D2-Net [14]
- DELF [42]
- Contextdesc [31]
- LFNet [43]
- R2D2 [47]
- Key.Net [5]
- DISK [62]
- ALIKED [6]
- Xfeat [7]
- KeyNetAffNetHardNet (KeyNet detector + AffNet + HardNet descriptor)

The following feature **descriptors** are supported:

- ORB [50]
- SIFT [28]
- ROOT SIFT
- SURF [8]
- AKAZE [2]
- BRISK [22]
- FREAK
- SuperPoint
- Tfeat
- BOOST-DESC [61]
- DAISY [60]

- LATCH [23]
- LUCID
- VGG [53]
- Hardnet [36]
- GeoDesc [65]
- SOSNet
- L2Net
- Log-polar descriptor
- D2-Net [14]
- DELF [42]
- Contextdesc [31]
- LFNet [43]
- R2D2 [47]
- BEBLID
- DISK [62]
- ALIKED [6]
- Xfeat [7]
- KeyNetAffNetHardNet (KeyNet detector + AffNet + HardNet descriptor)

For more information, refer to local_features/feature_types.py file. Some of the local features consist of a *joint detector-descriptor*. You can start playing with the supported local features by taking a look at `test/cv/test_feature_manager.py` and `main_feature_matching.py`.

In both the scripts `main_vo.py` and `main_slam.py`, you can create your preferred detector-descritor configuration and feed it to the function `feature_tracker_factory()`. Some ready-to-use configurations are already available in the file local_features/feature_tracker.configs.py

The function `feature_tracker_factory()` can be found in the file `local_features/feature_tracker.py`. Take a look at the file `local_features/feature_manager.py` for further details.

**N.B.**: You just need a *single* python environment to be able to work with all the supported local features!

## 4.2 Supported matchers

- `BF`: Brute force matcher on descriptors (with KNN).
- FLANN [38]
- XFeat [7]
- LightGlue
- LoFTR

See the file `local_features/feature_matcher.py` for further details.

## 4.3 Supported global descriptors and local descriptor aggregation methods

**Local descriptor aggregation methods**

- Bag of Words (BoW): DBoW2 [17], DBoW3. [paper]
- Vector of Locally Aggregated Descriptors: VLAD [3]. [paper]
- Incremental Bags of Binary Words (iBoW) via Online Binary Image Index: iBoW, OBIndex2. [paper]
- Hyperdimensional Computing: HDC [40]. [paper]

**NOTE**: *iBoW* and *OBIndex2* incrementally build a binary image index and do not need a prebuilt vocabulary. In the implemented classes, when needed, the input non-binary local descriptors are transparently transformed into binary descriptors.

### Global descriptors

Also referred to as *holistic descriptors*:

- SAD
- AlexNet
- NetVLAD [3]
- HDC-DELF
- CosPlace [9]
- EigenPlaces [10]

Different loop closing methods are available. These combines the above aggregation methods and global descriptors. See the file loop_closing/loop_detector_configs.py for further details.

## 4.4 Supported depth prediction models

Both monocular and stereo depth prediction models are available. SGBM algorithm has been included as a classic reference approach.

- SGBM: Depth SGBM from OpenCV (Stereo, classic approach) [18]
- Depth-Pro (Monocular) [11]
- DepthAnythingV2 (Monocular) [56]
- RAFT-Stereo (Stereo) [57]
- CREStereo (Stereo) [25]
- MASt3R (Monocular/Stereo) [21]
- MV-DUSt3R (Monocular/Stereo) [55]

## 4.5 Supported volumetric mapping methods

- TSDF with voxel block grid (parallel spatial hashing) [13]
- Incremental 3D Gaussian Splatting. See here and MonoGS for a description of its backend [34, 19].

## 4.6 Configuration

### Main configuration file

Refer to this section for how to update the main configuration file config.yaml and affect the configuration parameters in config_parameters.py.

### Datasets

The following datasets are supported:

| Dataset | type in `config.yaml` |
|---|---|
| KITTI odometry data set (grayscale, 22 GB) | `type: KITTI_DATASET` |
| TUM dataset | `type: TUM_DATASET` |
| ICL-NUIM dataset | `type: ICL_NUIM_DATASET` |
| EUROC dataset | `type: EUROC_DATASET` |
| REPLICA dataset | `type: REPLICA_DATASET` |
| TARTANAIR dataset | `type: TARTANAIR_DATASET` |
| ROS1 bags | `type: ROS1BAG_DATASET` |
| ROS2 bags | `type: ROS2BAG_DATASET` |
| Video file | `type: VIDEO_DATASET` |
| Folder of images | `type: FOLDER_DATASET` |

Use the download scripts available in the folder `scripts` to download some of the following datasets.

### KITTI Datasets

pySLAM code expects the following structure in the specified KITTI path folder (specified in the section `KITTI_DATASET` of the file `config.yaml`). :

```
+-- sequences
|   +-- 00
|   ...
|   +-- 21
```

```
+-- poses
    +-- 00.txt
    ...
    +-- 10.txt
```

1. Download the dataset (grayscale images) from http://www.cvlibs.net/datasets/kitti/eval_odometry.php and prepare the KITTI folder as specified above
2. Select the corresponding calibration settings file (section `KITTI_DATASET: settings:` in the file `config.yaml`)

### TUM Datasets

pySLAM code expects a file `associations.txt` in each TUM dataset folder (specified in the section `TUM_DATASET:` of the file `config.yaml`).

1. Download a sequence from http://vision.in.tum.de/data/datasets/rgbd-dataset/download and uncompress it.
2. Associate RGB images and depth images using the python script associate.py. You can generate your `associations.txt` file by executing: `bash $ python associate.py PATH_TO_SEQUENCE/rgb.txt PATH_TO_SEQUENCE/depth.txt > associations.txt # pay attention to the order!`
3. Select the corresponding calibration settings file (section `TUM_DATASET: settings:` in the file `config.yaml`).

### ICL-NUIM Datasets

Follow the same instructions provided for the TUM datasets.

### EuRoC Datasets

1. Download a sequence (ASL format) from http://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets (check this direct link)
2. Use the script `io/generate_euroc_groundtruths_as_tum.sh` to generate the TUM-like groundtruth files `path + '/' + name + '/mav0/state_groundtruth_estimate0/data.tum'` that are required by the `EurocGroundTruth` class.
3. Select the corresponding calibration settings file (section `EUROC_DATASET: settings:` in the file `config.yaml`).

### Replica Datasets

1. You can download the zip file containing all the sequences by running:
   `$ wget https://cvg-data.inf.ethz.ch/nice-slam/data/Replica.zip`
2. Then, uncompress it and deploy the files as you wish.
3. Select the corresponding calibration settings file (section `REPLICA_DATASET: settings:` in the file `config.yaml`).

### Tartanair Datasets

1. You can download the datasets from https://theairlab.org/tartanair-dataset/
2. Then, uncompress them and deploy the files as you wish.
3. Select the corresponding calibration settings file (section `TARTANAIR_DATASET: settings:` in the file `config.yaml`).

### ROS1 bags

1. Source the main ROS1 `setup.bash` after you have sourced the `pyslam` python environment.
2. Set the paths and `ROS1BAG_DATASET: ros_parameters` in the file `config.yaml`.
3. Select/prepare the correspoding calibration settings file (section `ROS1BAG_DATASET: settings:` in the file `config.yaml`). See the available yaml files in the folder `Settings` as an example.

### ROS2 bags

1. Source the main ROS2 `setup.bash` after you have sourced the `pyslam` python environment.
2. Set the paths and `ROS2BAG_DATASET: ros_parameters` in the file `config.yaml`.
3. Select/prepare the correspoding calibration settings file (section `ROS2BAG_DATASET: settings:` in the file `config.yaml`). See the available yaml files in the folder `Settings` as an example.

### Video and Folder Datasets

You can use the VIDEO_DATASET and FOLDER_DATASET types to read generic video files and image folders (specifying a glob pattern), respectively. A companion ground truth file can be set in the simple format type: Refer to the class SimpleGroundTruth in io/ground_truth.py and check the script io/convert_groundtruth_to_simple.py.

## 4.7 Camera Settings

The folder settings contains the camera settings files which can be used for testing the code. These are the same used in the framework ORB-SLAM2 [39]. You can easily modify one of those files for creating your own new calibration file (for your new datasets).

In order to calibrate your camera, you can use the scripts in the folder calibration. In particular: 1. Use the script grab_chessboard_images.py to collect a sequence of images where the chessboard can be detected (set the chessboard size therein, you can use the calibration pattern calib_pattern.pdf in the same folder) 2. Use the script calibrate.py to process the collected images and compute the calibration parameters (set the chessboard size therein)

For more information on the calibration process, see this tutorial [32] or this other link [45].

If you want to **use your camera**, you have to:

- Calibrate it and configure WEBCAM.yaml accordingly.
- Record a video (for instance, by using save_video.py in the folder calibration).
- Configure the VIDEO_DATASET section of config.yaml in order to point to your recorded video.

# 5 Credits

The following is a list of frameworks that inspired or has been integrated into pySLAM. Many thanks to their Authors for their great work.

- Pangolin
- g2opy
- ORBSLAM2 [39]
- SuperPointPretrainedNetwork [12]
- Tfeat [4]
- Image Matching Benchmark Baselines [64]
- Hardnet [37]
- GeoDesc [30]
- SOSNet [59]
- L2Net [58]
- Log-polar descriptor [16]
- D2-Net [15]
- DELF [41]
- Contextdesc [29]
- LFNet [44]
- R2D2 [48]
- BEBLID [54]
- DISK [63]
- Xfeat [46]
- LightGlue [26]
- Key.Net [5]
- Twitchslam
- MonoVO
- VPR_Tutorial [51]
- DepthAnythingV2 [66]
- DepthPro [11]
- RAFT-Stereo [27]
- CREStereo and CREStereo-Pytorch [24]
- MonoGS [35]
- MASt3R [21]
- MV-DUSt3R [55]
- Many thanks to Anathonic for adding the trajectory-saving feature and for the comparison notebook: pySLAM vs ORB-SLAM3.

## 6 Contributing to pySLAM

If you like pySLAM and would like to contribute to the code base, you can report bugs, leave comments and proposing new features through issues and pull requests on github. Feel free to get in touch at *luigifreda(at)gmail[dot]com*. Thank you!

## References


[1] Pablo F Alcantarilla, Adrien Bartoli, and Andrew J Davison. Kaze features. *European conference on computer vision*, pages 214–227, 2012.

[2] Pablo F Alcantarilla, Jesús Nuevo, and Adrien Bartoli. Fast explicit diffusion for accelerated features in nonlinear scale spaces. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1281–1298, 2013.

[3] Relja Arandjelovic, Petr Gronat, Akihiko Torii, Tomas Pajdla, and Josef Sivic. Netvlad: Cnn architecture for weakly supervised place recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5297–5307, 2016.

[4] Vassileios Balntas, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Learning local feature descriptors with triplets and shallow convolutional neural networks. In *Bmvc*, volume 1, page 3, 2016.

[5] Axel Barroso-Laguna, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Key.net: Keypoint detection by handcrafted and learned cnn filters. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5836–5844, 2020.

[6] Axel Barroso-Laguna, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Aliked: A lightweight keypoint detector and descriptor. *arXiv preprint arXiv:2304.03608*, 2023.

[7] Axel Barroso-Laguna, Edgar Riba, Daniel Ponsa, and Krystian Mikolajczyk. Xfeat: A new feature detector and descriptor. *arXiv preprint arXiv:2404.19174*, 2024.

[8] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. *European conference on computer vision*, pages 404–417, 2006.

[9] Gabriele Berton, Carlo Masone, and Barbara Caputo. Cosplace: Efficient place recognition with cosine similarity. *arXiv preprint arXiv:2304.03608*, 2023.

[10] Gabriele Berton, Carlo Masone, and Barbara Caputo. Eigenplaces: Learning place recognition with eigenvectors. *arXiv preprint arXiv:2404.19174*, 2023.

[11] Aleksei Bochkovskii, Amaël Delaunoy, Hugo Germain, Marcel Santos, Yichao Zhou, Stephan R Richter, and Vladlen Koltun. Depth pro: Sharp monocular metric depth in less than a second. *arXiv preprint arXiv:2410.02073*, 2024.

[12] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. In *CVPR Deep Learning for Visual SLAM Workshop*, 2018.

[13] Wei Dong, Yixing Lao, Michael Kaess, and Vladlen Koltun. Ash: A modern framework for parallel spatial hashing in 3d perception. *IEEE transactions on pattern analysis and machine intelligence*, 45(5):5417–5435, 2022.

[14] Mihai Dusmanu, Ignacio Rocco, Tomas Pajdla, Marc Pollefeys, Josef Sivic, Akihiko Torii, and Torsten Sattler. D2-net: A trainable cnn for joint description and detection of local features. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8092–8101, 2019.

[15] Mihai Dusmanu, Ignacio Rocco, Tomas Pajdla, Marc Pollefeys, Josef Sivic, Akihiko Torii, and Torsten Sattler. D2-Net: A Trainable CNN for Joint Detection and Description of Local Features. In *Proceedings of the 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[16] Patrick Ebel, Anastasiia Mishchuk, Kwang Moo Yi, Pascal Fua, and Eduard Trulls. Beyond Cartesian Representations for Local Descriptors. 2019.

[17] Dorian Galvez-Lopez and Juan D Tardos. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, 2012.

[18] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on pattern analysis and machine intelligence*, 30(2):328–341, 2007.

[19] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Trans. Graph.*, 42(4):139–1, 2023.

[20] Georg Klein and David Murray. Parallel tracking and mapping for small ar workspaces. In *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, pages 225–234, 2007.

[21] Vincent Leroy, Yohann Cabon, and Jérôme Revaud. Grounding image matching in 3d with mast3r, 2024.

[22] Stefan Leutenegger, Margarita Chli, and Roland Y Siegwart. Brisk: Binary robust invariant scalable keypoints. *2011 International conference on computer vision*, pages 2548–2555, 2011.

[23] Gil Levi, Tal Hassner, and Ronen Basri. The latch descriptor: Local binary patterns for image matching. *IEEE transactions on pattern analysis and machine intelligence*, 38(8):1622–1634, 2016.

[24] Jiankun Li, Peisen Wang, Pengfei Xiong, Tao Cai, Ziwei Yan, Lei Yang, Jiangyu Liu, Haoqiang Fan, and Shuaicheng Liu. Practical stereo matching via cascaded recurrent network with adaptive correlation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16263–16272, 2022.

[25] Zhengfa Li, Yuhua Liu, Tianwei Shen, Shuaicheng Chen, Lu Fang, and Long Quan. Crestereo: Cross-scale cost aggregation for stereo matching. *arXiv preprint arXiv:2203.11483*, 2022.

[26] Philipp Lindenberger, Paul-Edouard Sarlin, and Marc Pollefeys. Lightglue: Local feature matching at light speed. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 17627–17638, 2023.

[27] Lahav Lipson, Zachary Teed, and Jia Deng. Raft-stereo: Multilevel recurrent field transforms for stereo matching. In *International Conference on 3D Vision (3DV)*, 2021.

[28] David G Lowe. Object recognition from local scale-invariant features. *Proceedings of the seventh IEEE international conference on computer vision*, 2:1150–1157, 1999.

[29] Zixin Luo, Tianwei Shen, Lei Zhou, Jiahui Zhang, Yao Yao, Shiwei Li, Tian Fang, and Long Quan. Contextdesc: Local descriptor augmentation with cross-modality context. *Computer Vision and Pattern Recognition (CVPR)*, 2019.

[30] Zixin Luo, Tianwei Shen, Lei Zhou, Siyu Zhu, Runze Zhang, Yao Yao, Tian Fang, and Long Quan. Geodesc: Learning local descriptors by integrating geometry constraints. In *Proceedings of the European conference on computer vision (ECCV)*, pages 168–183, 2018.

[31] Zixin Luo, Lei Zhou, Xiang Bai, Alan Yuille, and Jimmy Ren. Contextdesc: Local descriptor augmentation with cross-modality context. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2527–2536, 2020.

[32] Satya Mallick. Camera calibration using opencv, 2016.

[33] Jiri Matas, Ondrej Chum, Martin Urban, and Tomas Pajdla. Robust wide-baseline stereo from maximally stable extremal regions. *Proceedings of the British Machine Vision Conference*, 1(502):384–393, 2002.

[34] H Matsuki, R Murai, PH Kelly, and AJ Davison. Gaussian splatting slam. arxiv. *arXiv preprint arXiv:2312.06741*, 2023.

[35] Hidenobu Matsuki, Riku Murai, Paul H. J. Kelly, and Andrew J. Davison. Gaussian Splatting SLAM. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024.

[36] Anastasiia Mishchuk, Dmytro Mishkin, Filip Radenovic, and Jiri Matas. Working hard to know your neighbor's margins: Local descriptor learning loss. *Advances in neural information processing systems*, 30, 2017.

[37] Anastasiya Mishchuk, Dmytro Mishkin, Filip Radenovic, and Jiri Matas. Working hard to know your neighbor's margins: Local descriptor learning loss. In *Proceedings of NeurIPS*, December 2017.

[38] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[39] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: An open-source slam system for monocular, stereo and rgb-d cameras, 2017.

[40] Peer Neubert and Peter Protzel. Hyperdimensional computing as a framework for systematic aggregation of image descriptors. *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9067–9076, 2021.

[41] Hyeonwoo Noh, Andre Araujo, Jack Sim, Tobias Weyand, and Bohyung Han. Large-scale image retrieval with attentive deep local features. In *Proceedings of the IEEE international conference on computer vision*, pages 3456–3465, 2017.

[42] Hyeonwoo Noh, Andre Araujo, Joonseok Sim, Tobias Weyand, and Bohyung Han. Large-scale image retrieval with attentive deep local features. *Proceedings of the IEEE international conference on computer vision*, pages 3456–3465, 2017.

[43] Yoshitaka Ono, Eduard Trulls, Pascal Fua, and Kwang Moo Yi. Lf-net: Learning local features from images. *Advances in neural information processing systems*, 31, 2018.

[44] Yuki Ono, Eduard Trulls, Pascal Fua, and Kwang Moo Yi. Lf-net: Learning local features from images. *Advances in neural information processing systems*, 31, 2018.

[45] OpenCV. Camera calibration, 2021.

[46] Guilherme Potje, Felipe Cadar, André Araujo, Renato Martins, and Erickson R Nascimento. Xfeat: Accelerated features for lightweight image matching. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2682–2691, 2024.

[47] Jerome Revaud, Philippe Weinzaepfel, Cedric R De Souza, Nicolas Pion, Gabriela Csurka, Yohann Cabon, and Martin Humenberger. R2d2: Repeatable and reliable detector and descriptor. *Advances in neural information processing systems*, 32, 2019.

[48] Jerome Revaud, Philippe Weinzaepfel, César Roberto de Souza, and Martin Humenberger. R2D2: repeatable and reliable detector and descriptor. In *NeurIPS*, 2019.

[49] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. *European conference on computer vision*, pages 430–443, 2006.

[50] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. *2011 International conference on computer vision*, pages 2564–2571, 2011.

[51] Stefan Schubert, Peer Neubert, Sourav Garg, Michael Milford, and Tobias Fischer. Visual place recognition: A tutorial. *IEEE Robotics & Automation Magazine*, 2023.

[52] Jianbo Shi and Carlo Tomasi. Good features to track. *1994 Proceedings of IEEE conference on computer vision and pattern recognition*, pages 593–600, 1994.

[53] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Learning local feature descriptors using convex optimisation. *IEEE transactions on pattern analysis and machine intelligence*, 36(8):1573–1585, 2014.

[54] Iago Suárez, Ghesn Sfeir, José M Buenaposada, and Luis Baumela. Beblid: Boosted efficient binary local image descriptor. *Pattern recognition letters*, 133:366–372, 2020.

[55] Zhenggang Tang, Yuchen Fan, Dilin Wang, Hongyu Xu, Rakesh Ranjan, Alexander Schwing, and Zhicheng Yan. Mv-dust3r+: Single-stage scene reconstruction from sparse views in 2 seconds, 2024.

[56] DepthAnything Team. Depthanythingv2: A monocular depth prediction model. *arXiv preprint arXiv:2406.09414*, 2024.

[57] Zachary Teed and Jia Deng. Raft-stereo: Recurrent all-pairs field transforms for stereo matching. *arXiv preprint arXiv:2109.07547*, 2021.

[58] Yurun Tian, Bin Fan, and Fuchao Wu. L2-net: Deep learning of discriminative patch descriptor in euclidean space. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 661–669, 2017.

[59] Yurun Tian, Xin Yu, Bin Fan, Fuchao Wu, Huub Heijnen, and Vassileios Balntas. Sosnet: Second order similarity regularization for local descriptor learning. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11016–11025, 2019.

[60] Engin Tola, Vincent Lepetit, and Pascal Fua. Daisy: An efficient dense descriptor applied to wide-baseline stereo. *IEEE transactions on pattern analysis and machine intelligence*, 32(5):815–830, 2010.

[61] Tomasz Trzcinski, Marios Christoudias, Pascal Fua, and Vincent Lepetit. Boosting binary keypoint descriptors. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2874–2881, 2013.

[62] Maciej Tyszkiewicz, Pascal Fua, and Eduard Trulls. Disk: Learning local features with policy gradient. *Advances in neural information processing systems*, 33:14254–14265, 2020.

[63] Michał Tyszkiewicz, Pascal Fua, and Eduard Trulls. Disk: Learning local features with policy gradient. *Advances in Neural Information Processing Systems*, 33:14254–14265, 2020.

[64] vcg uvic. Image matching benchmark baselines, 2020.

[65] Yannick Verdie, Kwang Moo Yi, Pascal Fua, and Vincent Lepetit. Tilde: A temporally invariant learned detector. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5279–5288, 2015.

[66] Lihe Yang, Bingyi Kang, Zilong Huang, Zhen Zhao, Xiaogang Xu, Jiashi Feng, and Hengshuang Zhao. Depth anything v2. *arXiv:2406.09414*, 2024.