OpenVINS

Generated by Doxygen 1.9.4

Chapter 1

OpenVINS

Welcome to the OpenVINS project! The OpenVINS project houses some core computer vision code along with a state-of-the art filter-based visual-inertial estimator. The core filter is an `Extended Kalman filter` which fuses inertial information with sparse visual feature tracks. These visual feature tracks are fused leveraging the `Multi-State Constraint Kalman Filter (MSCKF)` sliding window formulation which allows for 3D features to update the state estimate without directly estimating the feature states in the filter. Inspired by graph-based optimization systems, the included filter has modularity allowing for convenient covariance management with a proper type-based state system. Please take a look at the feature list below for full details on what the system supports.

- Github project page - `https://github.com/rpng/open_vins`
- Documentation - `https://docs.openvins.com/`
- Getting started guide - `https://docs.openvins.com/getting-started.html`
- Publication reference - `http://udel.edu/~pgeneva/downloads/papers/c10.pdf`

1.1 News / Events

- **December 13, 2021** - New YAML configuration system, ROS2 support, Docker images, robust static initialization based on disparity, internal logging system to reduce verbosity, image transport publishers, dynamic number of features support, and other small fixes. See v2.5 `PR#209` for details.
- **July 19, 2021** - Camera classes, masking support, alignment utility, and other small fixes. See v2.4 `PR#117` for details.
- **December 1, 2020** - Released improved memory management, active feature pointcloud publishing, limiting number of features in update to bound compute, and other small fixes. See v2.3 `PR#117` for details.
- **November 18, 2020** - Released groundtruth generation utility package, `vicon2gt` to enable creation of groundtruth trajectories in a motion capture room for evaulating VIO methods.
- **July 7, 2020** - Released zero velocity update for vehicle applications and direct initialization when standing still. See `PR#79` for details.

- **May 18, 2020** - Released secondary pose graph example repository `ov_secondary` based on `VINS-Fusion`. OpenVINS now publishes marginalized feature track, feature 3d position, and first camera intrinsics and extrinsics. See `PR#66` for details and discussion.
- **April 3, 2020** - Released `v2.0` update to the codebase with some key refactoring, ros-free building, improved dataset support, and single inverse depth feature representation. Please check out the `release page` for details.
- **January 21, 2020** - Our paper has been accepted for presentation in `ICRA 2020`. We look forward to seeing everybody there! We have also added links to a few videos of the system running on different datasets.
- **October 23, 2019** - OpenVINS placed first in the `IROS 2019 FPV Drone Racing VIO Competition`. We will be giving a short presentation at the `workshop` at 12:45pm in Macau on November 8th.
- **October 1, 2019** - We will be presenting at the `Visual-Inertial Navigation: Challenges and Applications` workshop at `IROS 2019`. The submitted workshop paper can be found at `this` link.
- **August 21, 2019** - Open sourced `ov_maplab` for interfacing OpenVINS with the `maplab` library.
- **August 15, 2019** - Initial release of OpenVINS repository and documentation website!

1.2 Project Features

- Sliding window visual-inertial MSCKF
- Modular covariance type system
- Comprehensive documentation and derivations
- Extendable visual-inertial simulator
  - On manifold SE(3) b-spline
  - Arbitrary number of cameras
  - Arbitrary sensor rate
  - Automatic feature generation
- Five different feature representations
  1. Global XYZ
  2. Global inverse depth
  3. Anchored XYZ
  4. Anchored inverse depth
  5. Anchored MSCKF inverse depth
  6. Anchored single inverse depth
- Calibration of sensor intrinsics and extrinsics
  - Camera to IMU transform
  - Camera to IMU time offset
  - Camera intrinsics
- Environmental SLAM feature
  - OpenCV ARUCO tag SLAM features

  - Sparse feature SLAM features
- Visual tracking support
  - Monocular camera
  - Stereo camera
  - Binocular camera
  - KLT or descriptor based
  - Masked tracking
- Static IMU initialization (sfm will be open sourced later)
- Zero velocity detection and updates
- Out of the box evaluation on EurocMav, TUM-VI, UZH-FPV, KAIST Urban and VIO datasets
- Extensive evaluation suite (ATE, RPE, NEES, RMSE, etc..)

1.3 Codebase Extensions

- **ov_secondary** - This is an example secondary thread which provides loop closure in a loosely coupled manner for OpenVINS. This is a modification of the code originally developed by the HKUST aerial robotics group and can be found in their VINS-Fusion repository. Here we stress that this is a loosely coupled method, thus no information is returned to the estimator to improve the underlying OpenVINS odometry. This codebase has been modified in a few key areas including: exposing more loop closure parameters, subscribing to camera intrinsics, simplifying configuration such that only topics need to be supplied, and some tweaks to the loop closure detection to improve frequency.
- **ov_maplab** - This codebase contains the interface wrapper for exporting visual-inertial runs from OpenVINS into the ViMap structure taken by maplab. The state estimates and raw images are appended to the ViMap as OpenVINS runs through a dataset. After completion of the dataset, features are re-extract and triangulate with maplab's feature system. This can be used to merge multi-session maps, or to perform a batch optimization after first running the data through OpenVINS. Some example have been provided along with a helper script to export trajectories into the standard groundtruth format.
- **vicon2gt** - This utility was created to generate groundtruth trajectories using a motion capture system (e.g. Vicon or OptiTrack) for use in evaluating visual-inertial estimation systems. Specifically we calculate the inertial IMU state (full 15 dof) at camera frequency rate and generate a groundtruth trajectory similar to those provided by the EurocMav datasets. Performs fusion of inertial and motion capture information and estimates all unknown spacial-temporal calibrations between the two sensors.

1.4 Demo Videos

1.5 Credit / Licensing

This code was written by the Robot Perception and Navigation Group (RPNG) at the University of Delaware. If you have any issues with the code please open an issue on our github page with relevant implementation details and references. For researchers that have leveraged or compared to this work, please cite the following:

```
@Conference{Geneva2020ICRA,
  Title      = {{OpenVINS}: A Research Platform for Visual-Inertial Estimation},
  Author     = {Patrick Geneva and Kevin Eckenhoff and Woosik Lee and Yulin Yang and Guoquan Huang},
  Booktitle  = {Proc. of the IEEE International Conference on Robotics and Automation},
  Year       = {2020},
  Address    = {Paris, France},
  Url        = {\url{https://github.com/rpng/open_vins}}
}
```

The codebase is licensed under the GNU General Public License v3 (GPL-3).

Chapter 2

Getting Started

Welcome to the OpenVINS project! The following guides will help new users through the downloading of the software and running on datasets that we support. Additionally, we provide information on how to get your own sensors running on our system and have a guide on how we perform calibration. Please feel free to open an issue if you find any missing or areas that could be clarified.

2.1 High-level overview

From a high level the system is build on a few key algorithms. At the center we have the ov_core which contains a lot of standard computer vision algorithms and utilities that anybody can use. Specifically it stores the following large components:

- Sparse feature visual tracking (KLT and descriptor-based)
- Fundamental math types used to represent states
- Initialization procedures
- Multi-sensor simulator that generates synthetic measurements

This ov_core library is used by the ov_msckf system which contains our filter-based estimator. Within this we have the state, its manager, type system, prediction, and update algorithms. We encourage users to look at the specific documentation for a detailed view of what we support. The ov_eval library has a bunch of evaluation methods and scripts that one can use to generate research results for publication.

2.2 Getting Started Guides

- Installation Guide — Installation guide for OpenVINS and dependencies
- Building with Docker — Installing with Docker instead of from source
- Simple Tutorial — Simple tutorial on getting OpenVINS running out of the box.
- Supported Datasets — Links to supported datasets and configuration files
- Sensor Calibration — Guide to how to calibration your own visual-inertial sensors.

2.3 Installation Guide

2.3.1 ROS Dependency

Our codebase is built on top of the `Robot Operating System (ROS)` and has been tested building on Ubuntu 16.04, 18.04, 20.04 systems with ROS Kinetic, Melodic, and Noetic. We also recommend installing the `catkin_tools` build for easy ROS building. All ROS installs include `OpenCV`, but if you need to build OpenCV from source ensure you build the contributed modules as we use Aruco feature extraction. See the `opencv_contrib` readme on how to configure your cmake command when you build the core OpenCV library. We have tested building with OpenCV 3.2, 3.3, 3.4, 4.2, and 4.5. Please see the official instructions to install ROS:

- `Ubuntu 16.04 ROS 1 Kinetic` (uses OpenCV 3.3)
- `Ubuntu 18.04 ROS 1 Melodic` (uses OpenCV 3.2)
- `Ubuntu 20.04 ROS 1 Noetic` (uses OpenCV 4.2)
- `Ubuntu 18.04 ROS 2 Dashing` (uses OpenCV 3.2)
- `Ubuntu 20.04 ROS 2 Galactic` (uses OpenCV 4.2)

We do support ROS-free builds, but don't recommend using this interface as we have limited support for it. You will need to ensure you have installed OpenCV and Eigen3 which are the only dependencies. If ROS is not found on the system, one can use command line options to run the simulation without any visualization or `cmake -DENABLE_ROS=OFF ...`. If you are using the ROS-free interface, you will need to properly construct the ov_msckf::VioManagerOptions struct with proper information and feed inertial and image data into the correct functions. The simulator can give you and example on how to do this.

2.3.1.1 ROS1 Install

To install we can perform the following:
```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
       /etc/apt/sources.list.d/ros-latest.list'
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
sudo apt-get update
export ROS1_DISTRO=noetic # kinetic=16.04, melodic=18.04, noetic=20.04
sudo apt-get install ros-$ROS1_DISTRO-desktop-full
sudo apt-get install libeigen3-dev python-catkin-tools # ubuntu 16.04, 18.04
sudo apt-get install libeigen3-dev python3-catkin-tools python3-osrf-pycommon # ubuntu 20.04
```

If you only have ROS1 on your system and are not cross installing ROS2, then you can run the following to append this to your bashrc file. Every time a terminal is open, thus will load the ROS1 environmental variables required to find all dependencies for building and system installed packages.
```
echo "source /opt/ros/$ROS1_DISTRO/setup.bash" » ~/.bashrc
source ~/.bashrc
```

Otherwise, if you want to also install ROS2, you must *NOT* have a global source. Instead we can have a nice helper command which can be used when we build a ROS1 workspace. Additionally, the `source_devel` command can be used when in your workspace root to source built packages. Once appended simply run `source_ros1` to load your ROS1 environmental variables.
```
echo "alias source_ros1=\"source /opt/ros/$ROS1_DISTRO/setup.bash\"" » ~/.bashrc
echo "alias source_devel=\"source devel/setup.bash\"" » ~/.bashrc
source ~/.bashrc
```

2.3.1.2 ROS2 Install

To install we can perform the following:

```
sudo apt update && sudo apt install curl gnupg lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key  -o
       /usr/share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg]
       http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list
       > /dev/null
sudo apt-get update
export ROS2_DISTRO=galactic # dashing=18.04, galactic=20.04
sudo apt install ros-$ROS2_DISTRO-desktop
sudo apt-get install ros-$ROS2_DISTRO-ros2bag ros-$ROS2_DISTRO-rosbag2* # rosbag utilities (seems to be
       separate)
```

If you only have ROS2 on your system and are not cross installing ROS1, then you can run the following to append this to your bashrc file. Every time a terminal is open, thus will load the ROS2 environmental variables required to find all dependencies for building and system installed packages.

```
echo "source /opt/ros/$ROS2_DISTRO/setup.bash" » ~/.bashrc
source ~/.bashrc
```

Otherwise, if you want to also install ROS1, you must *NOT* have a global source. Instead we can have a nice helper command which can be used when we build a ROS1 workspace. Additionally, the `source_install` command can be used when in your workspace root to source built packages. Once appended simply run `source_ros2` to load your ROS1 environmental variables.

```
echo "alias source_ros2=\"source /opt/ros/$ROS2_DISTRO/setup.bash\"" » ~/.bashrc
echo "alias source_install=\"source install/setup.bash\"" » ~/.bashrc
source ~/.bashrc
```

2.3.2 Upgrading CMake to Version 3.12

We use a newer version of `cmake` so you will probably need to upgrade your system if you are on an old ubuntu version. You can run the following which downloads a script and will install it for you.

```
wget https://cmake.org/files/v3.13/cmake-3.13.5-Linux-x86_64.sh
sudo mkdir /opt/cmake
sudo sh cmake-3.13.5-Linux-x86_64.sh --prefix=/opt/cmake --skip-license
sudo ln -s /opt/cmake/bin/cmake /usr/local/bin/cmake
cmake --version
```

2.3.3 Cloning the OpenVINS Project

Now that we have ROS installed we can setup a catkin workspace and build the project! If you did not install the catkin_tools build system, you should be able to build using the standard `catkin_make` command that is included with ROS. If you run into any problems please google search the issue first and if you are unable to find a solution please open an issue on our github page. After the build is successful please following the Simple Tutorial guide on getting a dataset and running the system.

There are additional options that users might be interested in. Configure these with `catkin build -D<option↩ _name>=OFF` or `cmake -D<option_name>=ON ..` in the ROS free case.

- `ENABLE_ROS` - (default ON) - Enable or disable building with ROS (if it is found)
- `ENABLE_ARUCO_TAGS` - (default ON) - Enable or disable aruco tag (disable if no contrib modules)
- `BUILD_OV_EVAL` - (default ON) - Enable or disable building of ov_eval
- `DISABLE_MATPLOTLIB` - (default OFF) - Disable or enable matplotlib plot scripts in ov_eval

```
mkdir -p ~/workspace/catkin_ws_ov/src/
cd ~/workspace/catkin_ws_ov/src/
git clone https://github.com/rpng/open_vins/
cd ..
catkin build # ROS1
colcon build # ROS2
colcon build --event-handlers console_cohesion+ # ROS2 with verbose output
```

2.3.4 Additional Evaluation Requirements

If you want to use the plotting utility wrapper of matplotlib-cpp to generate plots directly from running the cpp code in ov_eval you will need to make sure you have a valid Python 2.7 or 3 install of matplotlib. On ubuntu 16.04 you can do the following command which should take care of everything you need. If you can't link properly, make sure you can call it from Python normally (i.e. that your Python environment is not broken). You can disable this visualization if it is broken for you by passing the -DDISABLE_MATPLOTLIB=ON parameter to your catkin build. Additionally if you wish to record CPU and memory usage of the node, you will need to install the psutil library.

```
sudo apt-get install python2.7-dev python-matplotlib python-numpy python-psutil # for python2 systems
sudo apt-get install python3-dev python3-matplotlib python3-numpy python3-psutil python3-tk # for python3
       systems
catkin build -DDISABLE_MATPLOTLIB=OFF # build with viz (default)
catkin build -DDISABLE_MATPLOTLIB=ON # build without viz
```

2.3.5 OpenCV Dependency (from source)

We leverage OpenCV for this project which you can typically use the install from ROS. If the ROS version of cv_↩ bridge does not work (or are using non-ROS building), then you can try building OpenCV from source ensuring you include the contrib modules. One should make sure you can see some of the "contrib" (e.g. aruco) when you cmake to ensure you have linked to the contrib modules.

OpenCV Install

Try to first build with your system / ROS OpenCV. Only fall back onto this if it does not allow you to compile, or want a newer version!

```
git clone https://github.com/opencv/opencv/
git clone https://github.com/opencv/opencv_contrib/
mkdir opencv/build/
cd opencv/build/
cmake -DOPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules ..
make -j8
sudo make install
```

If you do not want to build the modules, you should also be able to do this (while it is not as well tested). The ArucoTag tracker depends on a non-free module in the contrib repository, thus this will need to be disabled. You can disable this with `catkin build -DENABLE_ARUCO_TAGS=OFF` or `cmake -DENABLE_ARUCO_TAGS=OFF ..` in your build folder.

2.4 Building with Docker

2.4.1 Installing Docker

This will differ on which operating system you have installed, this guide is for linux-based systems. Please take a look at the official Docker Get Docker guide. There is also a guide from ROS called getting started with ROS and Docker. On Ubuntu one should be able to do the following to get docker:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
       /usr/share/keyrings/docker-archive-keyring.gpg
echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg]
       https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee
       /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

From there we can install `NVIDIA Container Toolkit` to allow for the docker to use our GPU and for easy GUI pass through. You might also want to check out `this` blogpost for some more details.

```
distribution=$(. /etc/os-release;echo $ID$VERSION_ID) \
   && curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | sudo apt-key add - \
   && curl -s -L https://nvidia.github.io/nvidia-docker/$distribution/nvidia-docker.list | sudo tee
       /etc/apt/sources.list.d/nvidia-docker.list
sudo apt-get update
sudo apt-get install -y nvidia-docker2
sudo systemctl restart docker
sudo docker run --rm --gpus all nvidia/cuda:11.0-base nvidia-smi #to verify install
```

From this point we should be able to "test" that everything is working ok. First on the host machine we need to allow for x11 windows to connect.

```
xhost +
```

We can now run the following command which should open gazebo GUI on your main desktop window.

```
docker run -it --net=host --gpus all \
    --env="NVIDIA_DRIVER_CAPABILITIES=all" \
    --env="DISPLAY" \
    --env="QT_X11_NO_MITSHM=1" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    osrf/ros:noetic-desktop-full \
    bash -it -c "roslaunch gazebo_ros empty_world.launch"
```

Alternatively we can launch directly into a bash shell and run commands from in there. This basically gives you a terminal in the docker container.

```
docker run -it --net=host --gpus all \
    --env="NVIDIA_DRIVER_CAPABILITIES=all" \
    --env="DISPLAY" \
    --env="QT_X11_NO_MITSHM=1" \
    --volume="/tmp/.X11-unix:/tmp/.X11-unix:rw" \
    osrf/ros:noetic-desktop-full \
    bash
rviz # you should be able to launch rviz once in bash
```

2.4.2 Running OpenVINS with Docker

Clone the OpenVINS repository, build the container and then launch it. The `Dockerfile` will not build the repo by default, thus you will need to build the project. We have a few docker files for each version of ROS and operating system we support. In the following we will use the `Dockerfile_ros1_20_04` which is for a ROS1 install with a 20.04 system.

Use a Workspace Directory Mount

Here it is important to note that we are going to create a dedicated ROS *workspace* which will then be loaded into the workspace. Thus if you are going to develop packages alongside OpenVINS you would make sure you have cloned your source code into the same workspace. The workspace local folder will be mounted to `/catkin_ws/` in the docker, thus all changes from the host are mirrored.

```
mkdir -p ~/workspace/catkin_ws_ov/src
cd ~/workspace/catkin_ws_ov/src
git clone https://github.com/rpng/open_vins.git
cd open_vins
export VERSION=ros1_20_04 # which docker file version you want (ROS1 vs ROS2 and ubuntu version)
docker build -t ov_$VERSION -f Dockerfile_$VERSION .
```

If the dockerfile breaks, you can remove the image and reinstall using the following:

```
docker image list
docker image rm ov_ros1_20_04 --force
```

From here it is a good idea to create a nice helper command which will launch the docker and also pass the GUI to your host machine. Here you can append it to the bottom of the ∼/.bashrc so that we always have it on startup or just run the two commands on each restart

Directory Binding

You will need to specify *absolute directory paths* to the workspace and dataset folders on the host you want to bind. Bind mounts are used to ensure that the host directory is directly used and all edits made on the host are sync'ed with the docker container. See the docker bind mounts documentation. You can add and remove mounts from this command as you see the need.

```
nano ~/.bashrc # add to the bashrc file
xhost + &> /dev/null
export DOCKER_CATKINWS=/home/username/workspace/catkin_ws_ov
export DOCKER_DATASETS=/home/username/datasets
alias ov_docker="docker run -it --net=host --gpus all \
    --env=\"NVIDIA_DRIVER_CAPABILITIES=all\" --env=\"DISPLAY\" \
    --env=\"QT_X11_NO_MITSHM=1\" --volume=\"/tmp/.X11-unix:/tmp/.X11-unix:rw\" \
    --mount type=bind,source=$DOCKER_CATKINWS,target=/catkin_ws \
    --mount type=bind,source=$DOCKER_DATASETS,target=/datasets $1"
source ~/.bashrc # after you save and exit
```

Now we can launch RVIZ and also compile the OpenVINS codebase. From two different terminals on the host machine one can run the following (ROS 1):

```
ov_docker ov_ros1_20_04 roscore
ov_docker ov_ros1_20_04 rosrun rviz rviz -d /catkin_ws/src/open_vins/ov_msckf/launch/display.rviz
```

To actually get a bash environment that we can use to build and run things with we can do the following. Note that any install or changes to operating system variables will not persist, thus only edit within your workspace which is linked as a volume.

```
ov_docker ov_ros1_20_04 bash
```

Now once inside the docker with the bash shell we can build and launch an example simulation:

```
cd catkin_ws
catkin build
source devel/setup.bash
rosrun ov_eval plot_trajectories none src/open_vins/ov_data/sim/udel_gore.txt
roslaunch ov_msckf simulation.launch
```

And a version for ROS 2 we can do the following:

```
cd catkin_ws
colcon build --event-handlers console_cohesion+
source install/setup.bash
ros2 run ov_eval plot_trajectories none src/open_vins/ov_data/sim/udel_gore.txt
ros2 run ov_msckf run_simulation src/open_vins/config/rpng_sim/estimator_config.yaml
```

Real-time Performance

On my machine running inside of the docker container is not real-time in nature. I am not sure why this is the case if someone knows if something is setup incorrectly please open a github issue. Thus it is recommended to only use the "serial" nodes which allows for the same parameters to be used as when installing directly on an OS.

2.4.3 Using Jetbrains Clion and Docker

Jetbrains provides some instructions on their side and a youtube video. Basically, Clion needs to be configured to use an external compile service and this service needs to be exposed from the docker container. I still recommend users compile with `catkin build` directly in the docker, but this will allow for debugging and syntax insights.

- https://blog.jetbrains.com/clion/2020/01/using-docker-with-clion/
- https://www.youtube.com/watch?v=h69XLiMtCT8

After building the OpenVINS image (as above) we can do the following which will start a detached process in the docker. This process will allow us to connect Clion to it.

```
export DOCKER_CATKINWS=/home/username/workspace/catkin_ws_ov # NOTE: should already be set in your bashrc
export DOCKER_DATASETS=/home/username/datasets # NOTE: should already be set in your bashrc
docker run -d --cap-add sys_ptrace -p127.0.0.1:2222:22 \
    --mount type=bind,source=$DOCKER_CATKINWS,target=/catkin_ws \
    --mount type=bind,source=$DOCKER_DATASETS,target=/datasets \
    --name clion_remote_env ov_ros1_20_04
```

We can now change Clion to use the docker remote:

1. In short, you should add a new Toolchain entry in settings under Build, Execution, Deployment as a Remote Host type.
2. Click in the Credentials section and fill out the SSH credentials we set-up in the Dockerfile
    - Host: localhost
    - Port: 2222
    - Username: user
    - Password: password
    - CMake: /usr/local/bin/cmake
3. Make sure the found CMake is the custom one installed and not the system one (greater than 3.12)
4. Add a CMake profile that uses this toolchain and you're done.
5. Change build target to be this new CMake profile (optionally just edit / delete the default)

To add support for ROS you will need to manually set environmental variables in the CMake profile. These were generated by going into the ROS workspace, building a package, and then looking at `printenv` output. It should be under `Settings > Build,Execution,Deployment > CMake > (your profile) > Environment`. This might be a brittle method, but not sure what else to do... (also see `this` blog post). You will need to edit the ROS version (`noetic` is used below) to fit whatever docker container you are using.

```
LD_PATH_LIB=/catkin_ws/devel/lib:/opt/ros/noetic/lib;PYTHON_EXECUTABLE=/usr/bin/python3;PYTHON_INCLUDE_DIR=/usr/include/python3.8
```

When you build in Clion you should see in `docker stats` that the `clion_remote_env` is building the files and maxing out the CPU during this process. Clion should send the source files to the remote server and then on build should build and run it remotely within the docker container. A user might also want to edit `Build,Execution,Deployment > Deployment` settings to exclude certain folders from copying over. See this `jetbrains documentation page` for more details.

2.5 Simple Tutorial

This guide assumes that you have already built the project successfully and are now ready to run the program on some datasets. If you have not compiled the program yet please follow the Installation Guide guide. The first that we will download is a dataset to run the program on. In this tutorial we will run on the `EuRoC MAV Dataset` Burri et al. [2016] which provides monochrome stereo images at 20Hz with a MEMS ADIS16448 IMU at 200Hz.

`Download ROS 1 Bag Vicon Room 1 01 Easy`

`Download ROS 2 Bag Vicon Room 1 01 Easy`

All configuration information for the system is exposed to the user in the configuration file, and can be overridden in the launch file. We will create a launch file that will launch our MSCKF estimation node and feed the ROS bag into the system. One can take a look in the `launch` folder for more examples. For OpenVINS we need to define a series of files:

- `estimator_config.yaml` - Contains OpenVINS specific configuration files. Each of these can be overridden in the launch file.
- `kalibr_imu_chain.yaml` - IMU noise parameters and topic information based on the sensor in the dataset. This should be the same as Kalibr's (see IMU Intrinsic Calibration (Offline)).
- `kalibr_imucam_chain.yaml` - Camera to IMU transformation and camera intrinsics. This should be the same as Kalibr's (see Camera Intrinsic Calibration (Offline)).

2.5.1 ROS 1 Tutorial

The ROS1 system uses the `roslaunch` system to manage and launch nodes. These files can launch multiple nodes, and each node can their own set of parameters set. Consider the below launch file. We can see the main parameter that is being passed into the estimator is the `config_path` file which has all configuration for this specific dataset. Additionally, we can see that we are launching the `run_subscribe_msckf` ROS 1 node, and are going to be overriding the `use_stereo` and `max_cameras` with the specificed values. ROS parameters always have priority, and you should see in the console that they have been successfully overridden.

```
<launch>
    <!-- what config we are going to run (should match folder name) -->
    <arg name="verbosity"   default="INFO" /> <!-- ALL, DEBUG, INFO, WARNING, ERROR, SILENT -->
    <arg name="config"      default="euroc_mav" /> <!-- euroc_mav, tum_vi, rpng_aruco -->
    <arg name="config_path" default="$(find ov_msckf)/../config/$(arg config)/estimator_config.yaml" />
    <!-- MASTER NODE! -->
    <node name="run_subscribe_msckf" pkg="ov_msckf" type="run_subscribe_msckf" output="screen">
        <param name="verbosity"    type="string" value="$(arg verbosity)" />
        <param name="config_path"  type="string" value="$(arg config_path)" />
        <param name="use_stereo"   type="bool"   value="true" />
        <param name="max_cameras"  type="int"    value="2" />
    </node>
</launch>
```

Since the configuration file for the EurocMav dataset has already been created, we can simply do the following. Note it is good practice to run a `roscore` that stays active so that you do not need to relaunch rviz or other packages.

```
roscore # term 0
source devel/setup.bash # term 1
roslaunch ov_msckf subscribe.launch config:=euroc_mav
```

In another two terminals we can run the following. For RVIZ, one can open the `ov_msckf/launch/display.↩ rviz` configuration file. You should see the system publishing features and a state estimate.

```
rviz # term 2
rosbag play V1_01_easy.bag # term 3
```

2.5.2 ROS 2 Tutorial

For ROS 2, launch files and nodes have become a bit more combersom due to the removal of a centralized communication method. This both allows for more distributed systems, but causes a bit more on the developer to perform integration. The launch system is described in `this` design article. Consider the following launch file which does the same as the ROS 1 launch file above.

```
from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, LogInfo, OpaqueFunction
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, TextSubstitution
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory, get_package_prefix
import os
import sys
launch_args = [
    DeclareLaunchArgument(name='namespace',         default_value='',          description='namespace'),
    DeclareLaunchArgument(name='config',            default_value='euroc_mav',  description='euroc_mav, tum_vi,
       rpng_aruco...'),
    DeclareLaunchArgument(name='verbosity',         default_value='INFO',       description='ALL, DEBUG, INFO,
       WARNING, ERROR, SILENT'),
    DeclareLaunchArgument(name='use_stereo',        default_value='true',       description=''),
    DeclareLaunchArgument(name='max_cameras',       default_value='2',          description='')
]
def launch_setup(context):
    configs_dir=os.path.join(get_package_share_directory('ov_msckf'),'..','config')
    available_configs = os.listdir(configs_dir)
    config = LaunchConfiguration('config').perform(context)
    if not config in available_configs:
        return[LogInfo(msg='ERROR: unknown config: \'{}\' - Available configs are: {} - not starting
      OpenVINS'.format(config,', '.join(available_configs)))]
    config_path = os.path.join(get_package_share_directory('ov_msckf'),'config',config,'estimator_config.yaml')
    node1 = Node(package = 'ov_msckf',
                 executable = 'run_subscribe_msckf',
                 namespace = LaunchConfiguration('namespace'),
                 parameters =[{'verbosity': LaunchConfiguration('verbosity')},
                              {'use_stereo': LaunchConfiguration('use_stereo')},
                              {'max_cameras': LaunchConfiguration('max_cameras')},
                              {'config_path': config_path}])
    return [node1]
def generate_launch_description():
    opfunc = OpaqueFunction(function = launch_setup)
    ld = LaunchDescription(launch_args)
    ld.add_action(opfunc)
    return ld
```

We can see that first the `launch_setup` function defines the nodes that we will be launching from this file. Then the `LaunchDescription` is created given the launch arguments and the node is added to it and returned to ROS. We can the launch it using the following:

```
source install/setup.bash
ros2 launch ov_msckf subscribe.launch.py config:=euroc_mav
```

We can then use the ROS2 rosbag file. First make sure you have installed the rosbag2 and all its backends. If you downloaded the bag above you should already have a valid bag format. Otherwise, you will need to convert it following ROS1 to ROS2 Bag Conversion Guide . A "bag" is now defined by a db3 sqlite database and config yaml file in a folder. In another terminal we can run the following:

```
ros2 bag play V1_01_easy
```

2.6 Supported Datasets

2.6.1 The EuRoC MAV Dataset

The ETH ASL `EuRoC MAV dataset` Burri et al. [2016] is one of the most used datasets in the visual-inertial / simultaneous localization and mapping (SLAM) research literature. The reason for this is the synchronised inertial+camera sensor data and the high quality groundtruth. The dataset contains different sequences of varying difficulty of a Micro Aerial Vehicle (MAV) flying in an indoor room. Monochrome stereo images are collected by a two Aptina MT9V034 global shutter cameras at 20 frames per seconds, while a ADIS16448 MEMS inertial unit provides linear accelerations and angular velocities at a rate of 200 samples per second.

We recommend that most users start testing on this dataset before moving on to the other datasets that our system support or before trying with your own collected data. The machine hall datasets have the MAV being picked up in the beginning and then set down, we normally skip this part, but it should be able to be handled by the filter if SLAM features are enabled. Please take a look at the `run_ros_eth.sh` script for some reasonable default values (they might still need to be tuned).

Groundtruth on V1_01_easy

We have found that the groundtruth on the V1_01_easy dataset is not accurate in its orientation estimate. We have recomputed this by optimizing the inertial and vicon readings in a graph to get the trajectory of the imu. You can find the output at this `link` and is what we normally use to evaluate the error on this dataset.

Dataset Name	Length (m)	Dataset Link	Groundtruth Traj.	Config
Vicon Room 1 01	58	rosbag, rosbag2	link	config
Vicon Room 1 02	76	rosbag, rosbag2	link	config
Vicon Room 1 03	79	rosbag, rosbag2	link	config
Vicon Room 2 01	37	rosbag, rosbag2	link	config
Vicon Room 2 02	83	rosbag, rosbag2	link	config
Vicon Room 2 03	86	rosbag, rosbag2	link	config
Machine Hall 01	80	rosbag, rosbag2	link	config
Machine Hall 02	73	rosbag, rosbag2	link	config
Machine Hall 03	131	rosbag, rosbag2	link	config
Machine Hall 04	92	rosbag, rosbag2	link	config
Machine Hall 05	98	rosbag, rosbag2	link	config

2.6.2 TUM Visual-Inertial Dataset

The TUM `Visual-Inertial Dataset` Schubert et al. [2018] is a more recent dataset that was presented to provide a way to evaluate state-of-the-art visual inertial odometry approaches. As compared to the EuRoC MAV datasets, this dataset provides photometric calibration of the cameras which has not been available in any other visual-inertal dataset for researchers. Monochrome stereo images are collected by two IDS uEye UI-3241LE-M-GL global shutter cameras at 20 frames per second, while a Bosch BMI160 inertial unit provides linear accelerations and angular velocities at a rate of 200 samples per second. Not all datasets have groundtruth available throughout the entire trajectory as the motion capture system is limited to the starting and ending room. There are quite a few very challenging outdoor handheld datasets which are a challenging direction for research. Note that we focus on the room datasets as full 6 dof pose collection is available over the total trajectory.

Filter Initialization from Standstill

These datasets have very non-static starts, as they are handheld, and the standstill initialization has issues handling this. Thus careful tuning of the imu initialization threshold is typically needed to ensure that the initialized orientation and the zero velocity assumption are valid. Please take a look at the `run_ros_tumvi.sh` script for some reasonable default values (they might still need to be tuned).

Dataset Name	Length (m)	Dataset Link	Groundtruth Traj.	Config
room1	147	rosbag	link	config
room2	142	rosbag	link	config
room3	136	rosbag	link	config
room4	69	rosbag	link	config
room5	132	rosbag	link	config
room6	67	rosbag	link	config

2.6.3 RPNG OpenVINS Dataset

In additional the community maintained datasets, we have also released a few datasets. Please cite the OpenVINS paper if you use any of these datasets in your works. Here are the specifics of the sensors that each dataset uses:

- ArUco Datasets:

  - Core visual-inertial sensor is the `VI-Sensor`
  - Stereo global shutter images at 20 Hz
  - ADIS16448 IMU at 200 Hz
  - Kalibr calibration file can be found `here`

- Ironsides Datasets:
  - Core visual-inertial sensor is the `ironsides`
  - Has two `Reach RTK` one subscribed to a base station for corrections
  - Stereo global shutter fisheye images at 20 Hz
  - InvenSense IMU at 200 Hz
  - GPS fixes at 5 Hz (/reach01/tcpfix has corrections from `NYSNet`)
  - Kalibr calibration file can be found `here`

Most of these datasets do not have perfect calibration parameters, and some are not time synchronised. Thus, please ensure that you have enabled online calibration of these parameters. Additionally, there is no groundtruth for these datasets, but some do include GPS messages if you wish to compare relative to something.

Dataset Name	Length (m)	Dataset Link	Groundtruth Traj.	Config
ArUco Room 01	27	`rosbag`	none	`config aruco`
ArUco Room 02	93	`rosbag`	none	`config aruco`
ArUco Hallway 01	190	`rosbag`	none	`config aruco`
ArUco Hallway 02	105	`rosbag`	none	`config aruco`
Neighborhood 01	2300	`rosbag`	none	`config ironsides`
Neighborhood 02	7400	`rosbag`	none	`config ironsides`

2.6.4 UZH-FPV Drone Racing Dataset

The `UZH-FPV Drone Racing Dataset` Schubert et al. [2018] is a dataset focused on high-speed agressive 6dof motion with very high levels of optical flow as compared to other datasets. A FPV drone racing quadrotor has on board a Qualcomm Snapdragon Flight board which can provide inertial measurement and has two 640x480 grayscale global shutter fisheye camera's attached. The groundtruth is collected with a Leica Nova MS60 laser tracker. There are four total sensor configurations and calibration provides including: indoor forward facing stereo, indoor 45 degree stereo, outdoor forward facing, and outdoor 45 degree. A top speed of 12.8 m/s (28 mph) is reached in the indoor scenarios, and 23.4 m/s (54 mphs) is reached in the outdoor datasets. Each of these datasets is picked up in the beginning and then set down, we normally skip this part, but it should be able to be handled by the filter if SLAM features are enabled. Please take a look at the `run_ros_uzhfpv.sh` script for some reasonable default values (they might still need to be tuned).

Dataset Groundtruthing

Only the Absolute Trajectory Error (ATE) should be used as a metric for this dataset. This is due to inaccurate groundtruth orientation estimates which are explain in their `report` on the issue. The basic summary is that it is hard to get an accurate orientation information due to the point-based Leica measurements used to groundtruth.

Dataset Name	Length (m)	Dataset Link	Groundtruth Traj.	Config
Indoor 5	157	`rosbag`	`link`	`config`

Dataset Name	Length (m)	Dataset Link	Groundtruth Traj.	Config
Indoor 6	204	rosbag	link	config
Indoor 7	314	rosbag	link	config
Indoor 9	136	rosbag	link	config
Indoor 10	129	rosbag	link	config
Indoor 45deg 2	207	rosbag	link	config
Indoor 45deg 4	164	rosbag	link	config
Indoor 45deg 12	112	rosbag	link	config
Indoor 45deg 13	159	rosbag	link	config
Indoor 45deg 14	211	rosbag	link	config

2.6.5 KAIST Urban Dataset

The KAIST urban dataset Jeong et al. [2019] is a dataset focus on autonomous driving and localization in challenging complex urban environments. The dataset was collected in Korea with a vehicle equipped with stereo camera pair, 2d SICK LiDARs, 3d Velodyne LiDAR, Xsens IMU, fiber optic gyro (FoG), wheel encoders, and RKT GPS. The camera is 10 Hz, while the Xsens IMU is 100 Hz sensing rate. A groundtruth "baseline" trajectory is also provided which is the resulting output from fusion of the FoG, RKT GPS, and wheel encoders.

Dynamic Environments

A challenging open research question is being able to handle dynamic objects seen from the cameras. By default we rely on our tracking 8 point RANSAC to handle these dynamics objects. In the most of the KAIST datasets the majority of the scene can be taken up by other moving vehicles, thus the performance can suffer. Please be aware of this fact.

```
git clone https://github.com/rpng/file_player.git
git clone https://github.com/irapkaist/irp_sen_msgs.git
catkin build
rosrun file_player file_player
```

To use the dataset, the dataset's file player can be used to publish the sensor information on to ROS. See the above commands on what packages you need to clone into your ROS workspace. It is important to *disable* the "skip stop section" to ensure that we have continuous sensor feeds. Typically we process the datasets at 1.5x rate so we get a ∼20 Hz image feed and the datasets can be processed in a more efficient manor.

Dataset Name	Length (km)	Dataset Link	Groundtruth Traj.	Example Launch
Urban 28	11.47	download	link	config
Urban 38	11.42	download	link	config
Urban 39	11.06	download	link	config

2.6.6 KAIST VIO Dataset

The KAIST VIO dataset Jeon et al. [2021] is a dataset of a MAV in an indoor 3.15 x 3.60 x 2.50 meter environment which undergoes various trajectory motions. The camera is intel realsense D435i 25 Hz, while the IMU is 100 Hz sensing rate from the pixelhawk 4 unit. A groundtruth "baseline" trajectory is also provided from a OptiTrack Mocap system at 50 Hz, the bag files have the marker body frame to IMU frame already applied. This topic has been provided in ov_data for convinces sake.

Dataset Name	Length (km)	Dataset Link	Groundtruth Traj.	Example Launch
circle	29.99	download	link	config
circle_fast	64.15	download	link	config
circle_head	35.05	download	link	config
infinite	29.35	download	link	config
infinite_fast	54.24	download	link	config
infinite_head	37.45	download	link	config
rotation	7.82	download	link	config
rotation_fast	14.55	download	link	config
square	41.94	download	link	config
square_fast	44.07	download	link	config
square_head	50.00	download	link	config

2.7 Sensor Calibration

2.7.1 Visual-Inertial Sensors

One may ask why use a visual-inertial sensor? The main reason is because of the complimentary nature of the two different sensing modalities. The camera provides high density external measurements of the environment, while the IMU measures internal ego-motion of the sensor platform. The IMU is crucial in providing robustness to the estimator while also providing system scale in the case of a monocular camera.

However, there are some challenges when leveraging the IMU in estimation. An IMU requires estimating of additional bias terms and requires highly accurate calibration between the camera and IMU. Additionally small errors in the relative timestamps between the sensors can also degrade performance very quickly in dynamic trajectories. Within this *Open↩ VINS* project we address these by advocating the *online* estimation of these extrinsic and time offset parameters between the cameras and IMU.

2.7.2 Camera Intrinsic Calibration (Offline)

The first task is to calibrate the camera intrinsic values such as the focal length, camera center, and distortion coefficients. Our group often uses the `Kalibr` Furgale et al. [2013] calibration toolbox to perform both intrinsic and extrinsic offline calibrations, by proceeding the following steps:

1. Clone and build the `Kalibr` toolbox
2. Print out a calibration board to use (we normally use the `Aprilgrid 6x6 0.8x0.8 m (A0 page)`)
3. Ensure that your sensor driver is publishing onto ROS with correct timestamps.
4. Sensor preparations
   - Limit motion blur by decreasing exposure time
   - Publish at low framerate to allow for larger variance in dataset (2-5hz)
   - Ensure that your calibration board can be viewed in all areas of the image
   - Ensure that your sensor is in focus (can use their *kalibr_camera_focus* or just manually)

5. Record a ROS bag and ensure that the calibration board can be seen from different orientations, distances, and in each part of the image plane. You can either move the calibration board and keep the camera still or move the camera and keep the calibration board stationary.
6. Finally run the calibration
   - Use the kalibr_calibrate_cameras with your specified topic
   - Depending on amount of distortion, use the *pinhole-equi* for fisheye, or if a low distortion then use the *pinhole-radtan*
   - Depending on how many frames are in your dataset this can take on the upwards of a few hours.
7. Inspect the final result, pay close attention to the final reprojection error graphs, with a good calibration having less than $<$ 0.2-0.5 pixel reprojection errors.

2.7.3 IMU Intrinsic Calibration (Offline)

The other imporatnt intrinsic calibration is to compute the inertial sensor intrinsic noise characteristics, which are needed for the batch optimization to calibrate the camera to IMU transform and in any VINS estimator so that we can properly probabilistically fuse the images and inertial readings. Unfortunately, there is no mature open sourced toolbox to find these values, while one can try our `kalibr_allan` project, which is not optimized though. Specifically we are estimating the following noise parameters:

Parameter	YAML element	Symbol	Units
Gyroscope "white noise"	`gyroscope_noise_density`	σ_g	$\frac{rad}{s}\frac{1}{\sqrt{Hz}}$
Accelerometer "white noise"	`accelerometer_noise_density`	σ_a	$\frac{m}{s^2}\frac{1}{\sqrt{Hz}}$
Gyroscope "random walk"	`gyroscope_random_walk`	σ_{b_g}	$\frac{rad}{s^2}\frac{1}{\sqrt{Hz}}$
Accelerometer "random walk"	`accelerometer_random_walk`	σ_{b_a}	$\frac{m}{s^3}\frac{1}{\sqrt{Hz}}$

The standard way as explained in
[`IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Interferometric Fiber Optic Gyros` (page 71, section C)] is that we can compute an `Allan variance` plot of the sensor readings over different observation times (see below).

As shown in the above figure, if we compute the Allan variance we we can look at the value of a line at $\tau = 1$ with a -1/2 slope fitted to the left side of the plot to get the white noise of the sensor. Similarly, a line with 1/2 fitted to the right side can be evaluated at $\tau = 3$ to get the random walk noise. We have a package that can do this in matlab, but actual verification and conversion into a C++ codebase has yet to be done. Please refer to our [`kalibr_allan`] github project for details on how to generate this plot for your sensor and calculate the values. Note that one may need to inflate the calculated values by 10-20 times to get usable sensor values.

2.7.4 Dynamic IMU-Camera Calibration (Offline)

After obtaining the intrinsic calibration of both the camera and IMU, we can now perform dynamic calibration of the transform between the two sensors. For this we again leverage the [`Kalibr` calibration toolbox]. For these collected datasets, it is important to minimize the motion blur in the camera while also ensuring that you excite all axes of the IMU. One needs to have at least one translational motion along with two degrees of orientation change for these calibration parameters to be observable (please see our recent paper on why: [`Degenerate Motion Analysis for Aided INS With Online Spatial and Temporal Sensor Calibration`]). We recommend having as much change in orientation as possible in order to ensure convergence.

1. Clone and build the `Kalibr` toolbox
2. Print out a calibration board to use (we normally use the `Aprilgrid 6x6 0.8x0.8 m (A0 page)`)
3. Ensure that your sensor driver is publishing onto ROS with correct timestamps.
4. Sensor preparations
   - Limit motion blur by decreasing exposure time
   - Publish at high-ish framerate (20-30hz)
   - Publish your inertial reading at a reasonable rate (200-500hz)
5. Record a ROS bag and ensure that the calibration board can be seen from different orientations, distances, and mostly in the center of the image. You should move in *smooth* non-jerky motions with a trajectory that excites as many orientation and translational directions as possible at the same time. A 30-60 second dataset is normally enough to allow for calibration.
6. Finally run the calibration
   - Use the *kalibr_calibrate_imu_camera*
   - Input your static calibration file which will have the camera topics in it
   - You will need to make an `imu.yaml` file with your noise parameters.
   - Depending on how many frames are in your dataset this can take on the upwards of half a day.
7. Inspect the final result. You will want to make sure that the spline fitted to the inertial reading was properly fitted. Ensure that your estimated biases do not leave your 3-sigma bounds. If they do your trajectory was too dynamic, or your noise values are not good. Sanity check your final rotation and translation with hand-measured values.

Chapter 3

IMU Propagation Derivations

3.1 IMU Measurements

We use a 6-axis inertial measurement unit (IMU) to propagate the inertial navigation system (INS), which provides measurements of the local rotational velocity (angular rate) $\boldsymbol{\omega}_m$ and local translational acceleration $\mathbf{a}_m$:

$$\begin{aligned}\boldsymbol{\omega}_m(t) &= \boldsymbol{\omega}(t) + \mathbf{b}_g(t) + \mathbf{n}_g(t) \\ \mathbf{a}_m(t) &= \mathbf{a}(t) + {}^I_G\mathbf{R}(t)\,{}^G\mathbf{g} + \mathbf{b}_a(t) + \mathbf{n}_a(t)\end{aligned}$$

where $\boldsymbol{\omega}$ and $\mathbf{a}$ are the true rotational velocity and translational acceleration in the IMU local frame $\{I\}$, $\mathbf{b}_g$ and $\mathbf{b}_a$ are the gyroscope and accelerometer biases, and $\mathbf{n}_g$ $\mathbf{n}_a$ are white Gaussian noise, ${}^G\mathbf{g} = [0\ \ 0\ \ 9.81]^\top$ is the gravity expressed in the global frame $\{G\}$ (noting that the gravity is slightly different on different locations of the globe), and ${}^I_G\mathbf{R}$ is the rotation matrix from global to IMU local frame.

3.2 State Vector

We define our INS state vector $\mathbf{x}_I$ at time t as:

$$\mathbf{x}_I(t) = \begin{bmatrix} {}^I_G\bar{q}(t) \\ {}^G\mathbf{p}_I(t) \\ {}^G\mathbf{v}_I(t) \\ \mathbf{b}_\mathbf{g}(t) \\ \mathbf{b}_\mathbf{a}(t) \end{bmatrix}$$

where ${}^I_G\bar{q}$ is the unit quaternion representing the rotation global to IMU frame, ${}^G\mathbf{p}_I$ is the position of IMU in global frame, and ${}^G\mathbf{v}_I$ is the velocity of IMU in global frame. We will often write time as a subscript of I describing the state of IMU at the time for notation clarity (e.g., ${}^{I_t}_G\bar{q} = {}^I_G\bar{q}(t)$). In order to define the IMU error state, the standard additive error definition is employed for the position, velocity, and biases, while we use the quaternion error state $\delta\bar{q}$ with a left quaternion multiplicative error $\otimes$:

$$\begin{aligned}
{}^I_G\bar{q} &= \delta\bar{q} \otimes {}^I_G\hat{\bar{q}} \\
\delta\bar{q} &= \begin{bmatrix} \hat{\mathbf{k}}\sin(\frac{1}{2}\tilde{\theta}) \\ \cos(\frac{1}{2}\tilde{\theta}) \end{bmatrix} \simeq \begin{bmatrix} \frac{1}{2}\tilde{\boldsymbol{\theta}} \\ 1 \end{bmatrix}
\end{aligned}$$

where $\hat{\mathbf{k}}$ is the rotation axis and $\tilde{\theta}$ is the rotation angle. For small rotation, the error angle vector is approximated by $\tilde{\boldsymbol{\theta}} = \tilde{\theta}\,\hat{\mathbf{k}}$ as the error vector about the three orientation axes. The total IMU error state thus is defined as the following 15x1 (not 16x1) vector:

$$\tilde{\mathbf{x}}_I(t) = \begin{bmatrix} {}^I_G\tilde{\boldsymbol{\theta}}(t) \\ {}^G\tilde{\mathbf{p}}_I(t) \\ {}^G\tilde{\mathbf{v}}_I(t) \\ \tilde{\mathbf{b}}_g(t) \\ \tilde{\mathbf{b}}_a(t) \end{bmatrix}$$

3.3 IMU Kinematics

The IMU state evolves over time as follows (see Indirect Kalman Filter for 3D Attitude Estimation Trawny and Roumeliotis [2005]).

$$\begin{aligned}
{}^I_G\dot{\bar{q}}(t) &= \frac{1}{2}\begin{bmatrix} -\lfloor\boldsymbol{\omega}(t)\times\rfloor & \boldsymbol{\omega}(t) \\ -\boldsymbol{\omega}^\top(t) & 0 \end{bmatrix} {}^{I_t}_G\bar{q} \\
&=: \frac{1}{2}\boldsymbol{\Omega}(\boldsymbol{\omega}(t)){}^{I_t}_G\bar{q} \\
{}^G\dot{\mathbf{p}}_I(t) &= {}^G\mathbf{v}_I(t) \\
{}^G\dot{\mathbf{v}}_I(t) &= {}^{I_t}_G\mathbf{R}^\top\mathbf{a}(t) \\
\dot{\mathbf{b}}_{\mathbf{g}}(t) &= \mathbf{n}_{wg} \\
\dot{\mathbf{b}}_{\mathbf{a}}(t) &= \mathbf{n}_{wa}
\end{aligned}$$

where we have modeled the gyroscope and accelerometer biases as random walk and thus their time derivatives are white Gaussian. Note that the above kinematics have been defined in terms of the *true* acceleration and angular velocities.

3.4 Continuous-time IMU Propagation

Given the continuous-time measurements $\boldsymbol{\omega}_m(t)$ and $\mathbf{a}_m(t)$ in the time interval $t \in [t_k, t_{k+1}]$, and their estimates, i.e. after taking the expectation, $\hat{\boldsymbol{\omega}}(t) = \boldsymbol{\omega}_m(t) - \hat{\boldsymbol{b}}_g(t)$ and $\hat{\boldsymbol{a}}(t) = \boldsymbol{a}_m(t) - \hat{\boldsymbol{b}}_a(t) - {}^I_G\hat{\mathbf{R}}(t)^G\mathbf{g}$, we can define the solutions to the above IMU kinematics differential equation. The solution to the quaternion evolution has the following general form:

$$ {}^{I_t}_G\bar{q} = \boldsymbol{\Theta}(t, t_k)\,{}^{I_k}_G\bar{q} $$

Differentiating and reordering the terms yields the governing equation for $\boldsymbol{\Theta}(t, t_k)$ as

$$ \begin{aligned} \boldsymbol{\Theta}(t, t_k) &= {}^{I_t}_G\bar{q}\;{}^{I_k}_G\bar{q}^{-1} \\ \Rightarrow \dot{\boldsymbol{\Theta}}(t, t_k) &= {}^{I_t}_G\dot{\bar{q}}\;{}^{I_k}_G\bar{q}^{-1} \\ &= \frac{1}{2}\boldsymbol{\Omega}(\boldsymbol{\omega}(t))\;{}^{I_t}_G\bar{q}\;{}^{I_k}_G\bar{q}^{-1} \\ &= \frac{1}{2}\boldsymbol{\Omega}(\boldsymbol{\omega}(t))\boldsymbol{\Theta}(t, t_k) \end{aligned} $$

with $\boldsymbol{\Theta}(t_k, t_k) = \mathbf{I}_4$. If we take $\boldsymbol{\omega}(t) = \boldsymbol{\omega}$ to be constant over the the period $\Delta t = t_{k+1} - t_k$, then the above system is linear time-invarying (LTI), and $\boldsymbol{\Theta}$ can be solved as (see `[Stochastic Models, Estimation, and Control]` Maybeck [1982]):

$$ \begin{aligned} \boldsymbol{\Theta}(t_{k+1}, t_k) &= \exp\left(\frac{1}{2}\boldsymbol{\Omega}(\boldsymbol{\omega})\Delta t\right) \\ &= \cos\left(\frac{|\boldsymbol{\omega}|}{2}\Delta t\right)\cdot \mathbf{I}_4 + \frac{1}{|\boldsymbol{\omega}|}\sin\left(\frac{|\boldsymbol{\omega}|}{2}\Delta t\right)\cdot\boldsymbol{\Omega}(\boldsymbol{\omega}) \\ &\simeq \mathbf{I}_4 + \frac{\Delta t}{2}\boldsymbol{\Omega}(\boldsymbol{\omega}) \end{aligned} $$

where the approximation assumes small $|\boldsymbol{\omega}|$. We can formulate the quaternion propagation from t_k to t_{k+1} using the estimated rotational velocity $\hat{\boldsymbol{\omega}}(t) = \hat{\boldsymbol{\omega}}$ as:

$$ {}^{I_{k+1}}_G\hat{\bar{q}} = \exp\left(\frac{1}{2}\boldsymbol{\Omega}(\hat{\boldsymbol{\omega}})\Delta t\right){}^{I_k}_G\hat{\bar{q}} $$

Having defined the integration of the orientation, we can integrate the velocity and position over the measurement interval:

$$ \begin{aligned} {}^G\hat{\mathbf{v}}_{k+1} &= {}^G\hat{\mathbf{v}}_{I_k} + \int_{t_k}^{t_{k+1}} {}^G\hat{\mathbf{a}}(\tau)d\tau \\ &= {}^G\hat{\mathbf{v}}_{I_k} - {}^G\mathbf{g}\Delta t + \int_{t_k}^{t_{k+1}} {}^G_{I_\tau}\hat{\mathbf{R}}(\mathbf{a}_m(\tau) - \hat{\mathbf{b}}_{\mathbf{a}}(\tau))d\tau \end{aligned} $$

$$ \begin{aligned} {}^G\hat{\mathbf{p}}_{I_{k+1}} &= {}^G\hat{\mathbf{p}}_{I_k} + \int_{t_k}^{t_{k+1}} {}^G\hat{\mathbf{v}}_I(\tau)d\tau \\ &= {}^G\hat{\mathbf{p}}_{I_k} + {}^G\hat{\mathbf{v}}_{I_k}\Delta t - \frac{1}{2}{}^G\mathbf{g}\Delta t^2 + \int_{t_k}^{t_{k+1}}\int_{t_k}^{s} {}^G_{I_\tau}\hat{\mathbf{R}}(\mathbf{a}_m(\tau) - \hat{\mathbf{b}}_{\mathbf{a}}(\tau))d\tau ds \end{aligned} $$

Propagation of each bias $\hat{\mathbf{b}}_{\mathbf{g}}$ and $\hat{\mathbf{b}}_{\mathbf{a}}$ is given by:

$$\begin{aligned}
\hat{\mathbf{b}}_{\mathbf{g},k+1} &= \hat{\mathbf{b}}_{\mathbf{g},k} + \int_{t_{k+1}}^{t_k} \hat{\mathbf{n}}_{wg}(\tau) d\tau \\
&= \hat{\mathbf{b}}_{\mathbf{g},k} \\
\hat{\mathbf{b}}_{\mathbf{a},k+1} &= \hat{\mathbf{b}}_{\mathbf{a},k} + \int_{t_{k+1}}^{t_k} \hat{\mathbf{n}}_{wa}(\tau) d\tau \\
&= \hat{\mathbf{b}}_{\mathbf{a},k}
\end{aligned}$$

The biases will not evolve since our random walk noises $\hat{\mathbf{n}}_{wg}$ and $\hat{\mathbf{n}}_{wa}$ are zero-mean white Gaussian. All of the above integrals could be analytically or numerically solved if one wishes to use the continuous-time measurement evolution model.

3.5 Discrete-time IMU Propagation

A simpler method is to model the measurements as discrete-time over the integration period. To do this, the measurements can be assumed to be constant during the sampling period. We employ this assumption and approximate that the measurement at time t_k remains the same until we get the next measurement at t_{k+1}. For the quaternion propagation, it is the same as continuous-time propagation with constant measurement assumption $\boldsymbol{\omega}_m(t_k) = \boldsymbol{\omega}_{m,k}$. We use subscript k to denote it is the measurement we get at time t_k. Therefore the propagation of quaternion can be written as:

$$ {}^{I_{k+1}}_{G}\hat{\bar{q}} = \exp\left(\frac{1}{2}\boldsymbol{\Omega}\big(\boldsymbol{\omega}_{m,k} - \hat{\mathbf{b}}_{g,k}\big)\Delta t\right) {}^{I_k}_{G}\hat{\bar{q}} $$

For the velocity and position propagation we have constant $\mathbf{a}_m(t_k) = \mathbf{a}_{m,k}$ over $t \in [t_k, t_{k+1}]$. We can therefore directly solve for the new states as:

$$\begin{aligned}
{}^G\hat{\mathbf{v}}_{k+1} &= {}^G\hat{\mathbf{v}}_{I_k} - {}^G\mathbf{g}\Delta t + {}^{I_k}_{G}\hat{\mathbf{R}}^\top(\mathbf{a}_{m,k} - \hat{\mathbf{b}}_{\mathbf{a},k})\Delta t \\
{}^G\hat{\mathbf{p}}_{I_{k+1}} &= {}^G\hat{\mathbf{p}}_{I_k} + {}^G\hat{\mathbf{v}}_{I_k}\Delta t - \frac{1}{2}{}^G\mathbf{g}\Delta t^2 + \frac{1}{2}{}^{I_k}_{G}\hat{\mathbf{R}}^\top(\mathbf{a}_{m,k} - \hat{\mathbf{b}}_{\mathbf{a},k})\Delta t^2
\end{aligned}$$

The propagation of each bias is likewise the continuous system:

$$\begin{aligned}
\hat{\mathbf{b}}_{\mathbf{g},k+1} &= \hat{\mathbf{b}}_{\mathbf{g},k} \\
\hat{\mathbf{b}}_{\mathbf{a},k+1} &= \hat{\mathbf{b}}_{\mathbf{a},k}
\end{aligned}$$

3.6 Discrete-time Error-state Propagation

In order to propagate the covariance matrix, we should derive the error-state propagation, i.e., computing the system Jacobian $\mathbf{\Phi}(t_{k+1}, t_k)$ and noise Jacobian $\mathbf{G}_k$. In particular, when the covariance matrix of the continuous-time measurement noises is given by $\mathbf{Q}_c$, then the discrete-time noise covariance $\mathbf{Q}_d$ can be computed as (see [`Indirect Kalman Filter for 3D Attitude Estimation`] Trawny and Roumeliotis [2005] Eq. (129) and (130)):

$$\begin{aligned}
\sigma_g &= \frac{1}{\sqrt{\Delta t}}\,\sigma_{g_c} \\
\sigma_{bg} &= \sqrt{\Delta t}\,\sigma_{bg_c} \\
\mathbf{Q}_{meas} &= \begin{bmatrix} \frac{1}{\Delta t}\,\sigma_{g_c}^2\,\mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \frac{1}{\Delta t}\,\sigma_{a_c}^2\,\mathbf{I}_3 \end{bmatrix} \\
\mathbf{Q}_{bias} &= \begin{bmatrix} \Delta t\,\sigma_{bg_c}^2\,\mathbf{I}_3 & \mathbf{0}_3 \\ \mathbf{0}_3 & \Delta t\,\sigma_{ba_c}^2\,\mathbf{I}_3 \end{bmatrix}
\end{aligned}$$

where $\mathbf{n} = [\mathbf{n}_g\ \mathbf{n}_a\ \mathbf{n}_{bg}\ \mathbf{n}_{ba}]^\top$ are the discrete IMU sensor noises which have been converted from their continuous representations. We define the stacked discrete measurement noise as follows:

$$\mathbf{Q}_d = \begin{bmatrix} \mathbf{Q}_{meas} & \mathbf{0}_3 \\ \mathbf{0}_3 & \mathbf{Q}_{bias} \end{bmatrix}$$

The method of computing Jacobians is to "perturb" each variable in the system and see how the old error "perturbation" relates to the new error state. That is, $\mathbf{\Phi}(t_{k+1}, t_k)$ and $\mathbf{G}_k$ can be found by perturbing each variable as:

$$\tilde{\mathbf{x}}_I(t_{k+1}) = \mathbf{\Phi}(t_{k+1}, t_k)\tilde{\mathbf{x}}_I(t_k) + \mathbf{G}_k\mathbf{n}$$

For the orientation error propagation, we start with the $\mathbf{SO}(3)$ perturbation using ${}^I_G\mathbf{R} \approx (\mathbf{I}_3 - \lfloor {}^I_G\tilde{\boldsymbol{\theta}}\times\rfloor){}^I_G\hat{\mathbf{R}}$:

$$\begin{aligned}
{}^{I_{k+1}}_{G}\mathbf{R} &= {}^{I_{k+1}}_{I_k}\mathbf{R}\,{}^{I_k}_{G}\mathbf{R} \\
(\mathbf{I}_3 - \lfloor {}^{I_{k+1}}_{G}\tilde{\boldsymbol{\theta}}\times\rfloor){}^{I_{k+1}}_{G}\hat{\mathbf{R}} &\approx \exp(-{}^{I_k}\hat{\boldsymbol{\omega}}\Delta t - {}^{I_k}\tilde{\boldsymbol{\omega}}\Delta t)(\mathbf{I}_3 - \lfloor {}^{I_k}_{G}\tilde{\boldsymbol{\theta}}\times\rfloor){}^{I_k}_{G}\hat{\mathbf{R}} \\
&= \exp(-{}^{I_k}\hat{\boldsymbol{\omega}}\Delta t)\exp(-\mathbf{J}_r(-{}^{I_k}\hat{\boldsymbol{\omega}}\Delta t){}^{I_k}\tilde{\boldsymbol{\omega}}\Delta t)(\mathbf{I}_3 - \lfloor {}^{I_k}_{G}\tilde{\boldsymbol{\theta}}\times\rfloor){}^{I_k}_{G}\hat{\mathbf{R}} \\
&= {}^{I_{k+1}}_{I_k}\hat{\mathbf{R}}(\mathbf{I}_3 - \lfloor \mathbf{J}_r(-{}^{I_k}\hat{\boldsymbol{\omega}}\Delta t)\tilde{\boldsymbol{\omega}}_k\Delta t\times\rfloor)(\mathbf{I}_3 - \lfloor {}^{I_k}_{G}\tilde{\boldsymbol{\theta}}\times\rfloor){}^{I_k}_{G}\hat{\mathbf{R}}
\end{aligned}$$

where $\tilde{\boldsymbol{\omega}} = \boldsymbol{\omega} - \hat{\boldsymbol{\omega}} = -(\tilde{\mathbf{b}}_{\mathbf{g}} + \mathbf{n}_g)$ handles both the perturbation to the bias and measurement noise. $\mathbf{J}_r(\boldsymbol{\theta})$ is the right Jacobian of $\mathbf{SO}(3)$ that maps the variation of rotation angle in the parameter vector space into the variation in the tangent vector space to the manifold [see ov_core::Jr_so3()]. By neglecting the second order terms from above, we obtain the following orientation error propagation:

$${}^{I_{k+1}}_{G}\tilde{\boldsymbol{\theta}} \approx {}^{I_{k+1}}_{I_k}\hat{\mathbf{R}}\,{}^{I_k}_{G}\tilde{\boldsymbol{\theta}} - {}^{I_{k+1}}_{I_k}\hat{\mathbf{R}}\mathbf{J}_r({}^{I_{k+1}}_{I_k}\hat{\boldsymbol{\theta}})\Delta t(\tilde{\mathbf{b}}_{\mathbf{g},k} + \mathbf{n}_{\mathbf{g},k})$$

Now we can do error propagation of position and velocity using the same scheme:

$$
\begin{aligned}
{}^G\mathbf{p}_{I_{k+1}} &= {}^G\mathbf{p}_{I_k} + {}^G\mathbf{v}_{I_k}\Delta t - \frac{1}{2}{}^G\mathbf{g}\Delta t^2 + \frac{1}{2}{}^{I_k}_G\mathbf{R}^\top\mathbf{a}_k\Delta t^2 \\
{}^G\hat{\mathbf{p}}_{I_{k+1}} + {}^G\tilde{\mathbf{p}}_{I_{k+1}} &\approx {}^G\hat{\mathbf{p}}_{I_k} + {}^G\tilde{\mathbf{p}}_{I_k} + {}^G\hat{\mathbf{v}}_{I_k}\Delta t + {}^G\tilde{\mathbf{v}}_{I_k}\Delta t - \frac{1}{2}{}^G\mathbf{g}\Delta t^2 \\
&\quad + \frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top(\mathbf{I}_3 + \lfloor {}^{I_k}_G\tilde{\boldsymbol{\theta}}\times\rfloor)(\hat{\mathbf{a}}_k + \tilde{\mathbf{a}}_k)\Delta t^2
\end{aligned}
$$

$$
\begin{aligned}
{}^G\mathbf{v}_{k+1} &= {}^G\mathbf{v}_{I_k} - {}^G\mathbf{g}\Delta t + {}^{I_k}_G\mathbf{R}^\top\mathbf{a}_k\Delta t \\
{}^G\hat{\mathbf{v}}_{k+1} +{}^G\tilde{\mathbf{v}}_{k+1} &\approx {}^G\hat{\mathbf{v}}_{I_k} + {}^G\tilde{\mathbf{v}}_{I_k} - {}^G\mathbf{g}\Delta t + {}^{I_k}_G\hat{\mathbf{R}}^\top(\mathbf{I}_3 + \lfloor {}^{I_k}_G\tilde{\boldsymbol{\theta}}\times\rfloor)(\hat{\mathbf{a}}_k + \tilde{\mathbf{a}}_k)\Delta t
\end{aligned}
$$

where $\tilde{\mathbf{a}} = \mathbf{a} - \hat{\mathbf{a}} = -(\tilde{\mathbf{b}}_\mathbf{a} + \mathbf{n}_\mathbf{a})$. By neglecting the second order error terms, we obtain the following position and velocity error propagation:

$$
\begin{aligned}
{}^G\tilde{\mathbf{p}}_{I_{k+1}} &= {}^G\tilde{\mathbf{p}}_{I_k} + \Delta t^G\tilde{\mathbf{v}}_{I_k} - \frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top\lfloor\hat{\mathbf{a}}_k\Delta t^2\times\rfloor{}^{I_k}_G\tilde{\boldsymbol{\theta}} - \frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t^2(\tilde{\mathbf{b}}_{\mathbf{a},k} + \mathbf{n}_{\mathbf{a},k}) \\
{}^G\tilde{\mathbf{v}}_{k+1} &= {}^G\tilde{\mathbf{v}}_{I_k} - {}^{I_k}_G\hat{\mathbf{R}}^\top\lfloor\hat{\mathbf{a}}_k\Delta t\times\rfloor{}^{I_k}_G\tilde{\boldsymbol{\theta}} - {}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t(\tilde{\mathbf{b}}_{\mathbf{a},k} + \mathbf{n}_{\mathbf{a},k})
\end{aligned}
$$

The propagation of the two random-walk biases are as follows:

$$
\begin{aligned}
\mathbf{b}_{\mathbf{g},k+1} &= \mathbf{b}_{\mathbf{g},k} + \mathbf{n}_{wg} \\
\hat{\mathbf{b}}_{\mathbf{g},k+1} + \tilde{\mathbf{b}}_{\mathbf{g},k+1} &= \hat{\mathbf{b}}_{\mathbf{g},k} + \tilde{\mathbf{b}}_{\mathbf{g},k} + \mathbf{n}_{wg} \\
\tilde{\mathbf{b}}_{\mathbf{g},k+1} &= \tilde{\mathbf{b}}_{\mathbf{g},k} + \mathbf{n}_{wg}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{b}_{\mathbf{a},k+1} &= \mathbf{b}_{\mathbf{a},k} + \mathbf{n}_{wa} \\
\hat{\mathbf{b}}_{\mathbf{a},k+1} + \tilde{\mathbf{b}}_{\mathbf{a},k+1} &= \hat{\mathbf{b}}_{\mathbf{a},k} + \tilde{\mathbf{b}}_{\mathbf{a},k} + \mathbf{n}_{wa} \\
\tilde{\mathbf{b}}_{\mathbf{a},k+1} &= \tilde{\mathbf{b}}_{\mathbf{a},k} + \mathbf{n}_{wa}
\end{aligned}
$$

By collecting all the perturbation results, we can build $\boldsymbol{\Phi}(t_{k+1}, t_k)$ and $\mathbf{G}_k$ matrices as:

$$
\boldsymbol{\Phi}(t_{k+1}, t_k) = \begin{bmatrix}
{}^{I_{k+1}}_{I_k}\hat{\mathbf{R}} & \mathbf{0}_3 & \mathbf{0}_3 & -{}^{I_{k+1}}_{I_k}\hat{\mathbf{R}}\mathbf{J}_r({}^{I_{k+1}}_{I_k}\hat{\boldsymbol{\theta}})\Delta t & \mathbf{0}_3 \\
-\frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top\lfloor\hat{\mathbf{a}}_k\Delta t^2\times\rfloor & \mathbf{I}_3 & \Delta t\mathbf{I}_3 & \mathbf{0}_3 & -\frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t^2 \\
-{}^{I_k}_G\hat{\mathbf{R}}^\top\lfloor\hat{\mathbf{a}}_k\Delta t\times\rfloor & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 & -{}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t \\
\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 \\
\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3
\end{bmatrix}
$$

$$
\mathbf{G}_k = \begin{bmatrix}
-{}^{I_{k+1}}_{I_k}\hat{\mathbf{R}}\mathbf{J}_r({}^{I_{k+1}}_{I_k}\hat{\boldsymbol{\theta}})\Delta t & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 \\
\mathbf{0}_3 & -\frac{1}{2}{}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t^2 & \mathbf{0}_3 & \mathbf{0}_3 \\
\mathbf{0}_3 & -{}^{I_k}_G\hat{\mathbf{R}}^\top\Delta t & \mathbf{0}_3 & \mathbf{0}_3 \\
\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3 & \mathbf{0}_3 \\
\mathbf{0}_3 & \mathbf{0}_3 & \mathbf{0}_3 & \mathbf{I}_3
\end{bmatrix}
$$

Now, with the computed $\boldsymbol{\Phi}(t_{k+1}, t_k)$ and $\mathbf{G}_k$ matrices, we can propagate the covariance from t_k to t_{k+1}:

$$
\mathbf{P}_{k+1|k} = \boldsymbol{\Phi}(t_{k+1}, t_k)\mathbf{P}_{k|k}\boldsymbol{\Phi}(t_{k+1}, t_k)^\top + \mathbf{G}_k\mathbf{Q}_d\mathbf{G}_k^\top
$$

Chapter 4

First-Estimate Jacobian Estimators

4.1 EKF Linearized Error-State System

When developing an extended Kalman filter (EKF), one needs to linearize the nonlinear motion and measurement models about some linearization point. This linearization is one of the sources of error causing inaccuracies in the estimates (in addition to, for exmaple, model errors and measurement noise). Let us consider the following linearized error-state visual-inertial system:

$$\begin{aligned}\tilde{\mathbf{x}}_{k|k-1} &= \mathbf{\Phi}_{(k,k-1)}\,\tilde{\mathbf{x}}_{k-1|k-1} + \mathbf{G}_k\mathbf{w}_k \\ \tilde{\mathbf{z}}_k &= \mathbf{H}_k\,\tilde{\mathbf{x}}_{k|k-1} + \mathbf{n}_k\end{aligned}$$

where the state contains the inertial navigation state and a single environmental feature (noting that we do not include biases to simplify the derivations):

$$\mathbf{x}_k = \begin{bmatrix} {}^{I_k}_{G}\bar{q}^{\top} & {}^{G}\mathbf{p}_{I_k}^{\top} & {}^{G}\mathbf{v}_{I_k}^{\top} & {}^{G}\mathbf{p}_f^{\top}\end{bmatrix}^{\top}$$

Note that we use the left quaternion error state (see [Indirect Kalman Filter for 3D Attitude Estimation] Trawny and Roumeliotis [2005] for details). For simplicity we assume that the camera and IMU frame have an identity transform. We can compute the measurement Jacobian of a given feature based on the perspective projection camera model at the *k*-th timestep as follows:

$$\begin{aligned}\mathbf{H}_k &= \mathbf{H}_{proj,k}\,\mathbf{H}_{state,k} \\ &= \begin{bmatrix}\frac{1}{{}^{I}z} & 0 & \frac{-{}^{I}x}{({}^{I}z)^2} \\ 0 & \frac{1}{{}^{I}z} & \frac{-{}^{I}y}{({}^{I}z)^2}\end{bmatrix}\begin{bmatrix}\lfloor {}^{I_k}_{G}\mathbf{R}({}^{G}\mathbf{p}_f - {}^{G}\mathbf{p}_{I_k})\times\rfloor & -{}^{I_k}_{G}\mathbf{R} & \mathbf{0}_{3\times3} & {}^{I_k}_{G}\mathbf{R}\end{bmatrix} \\ &= \mathbf{H}_{proj,k}\,{}^{I_k}_{G}\mathbf{R}\begin{bmatrix}\lfloor({}^{G}\mathbf{p}_f - {}^{G}\mathbf{p}_{I_k})\times\rfloor {}^{I_k}_{G}\mathbf{R}^{\top} & -\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3}\end{bmatrix}\end{aligned}$$

The state-transition (or system Jacobian) matrix from timestep *k-1* to *k* as (see [IMU Propagation Derivations] for more details):

$$\mathbf{\Phi}_{(k,k-1)} = \begin{bmatrix} {}^{I_k}_{I_{k-1}}\mathbf{R} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} \\ -{}^{I_{k-1}}_{G}\mathbf{R}^\top \lfloor \boldsymbol{\alpha}(k,k-1)\times\rfloor & \mathbf{I}_{3\times3} & (t_k - t_{k-1})\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \\ -{}^{I_{k-1}}_{G}\mathbf{R}^\top \lfloor \boldsymbol{\beta}(k,k-1)\times\rfloor & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} \end{bmatrix}$$

$$\begin{aligned} \boldsymbol{\alpha}(k,k-1) &= \int_{t_{k-1}}^{k}\int_{t_{k-1}}^{s} {}^{I_{k-1}}_{\tau}\mathbf{R}(\mathbf{a}(\tau) - \mathbf{b}_a - \mathbf{w}_a)d\tau ds \\ \boldsymbol{\beta}(k,k-1) &= \int_{t_{k-1}}^{t_k} {}^{I_{k-1}}_{\tau}\mathbf{R}(\mathbf{a}(\tau) - \mathbf{b}_a - \mathbf{w}_a)d\tau \end{aligned}$$

where $\mathbf{a}(\tau)$ is the true acceleration at time τ, ${}^{I_k}_{I_{k-1}}\mathbf{R}$ is computed using the gyroscope angular velocity measurements, and ${}^G\mathbf{g} = [0\ 0\ 9.81]^\top$ is gravity in the global frame of reference. During propagation one would need to solve these integrals using either analytical or numerical integration, while we here are interested in how the state evolves in order to examine its observability.

4.2 Linearized System Observability

The observability matrix of this linearized system is defined by:

$$\mathcal{O} = \begin{bmatrix} \mathbf{H}_0\mathbf{\Phi}_{(0,0)} \\ \mathbf{H}_1\mathbf{\Phi}_{(1,0)} \\ \mathbf{H}_2\mathbf{\Phi}_{(2,0)} \\ \vdots \\ \mathbf{H}_k\mathbf{\Phi}_{(k,0)} \end{bmatrix}$$

where $\mathbf{H}_k$ is the measurement Jacobian at timestep *k* and $\mathbf{\Phi}_{(k,0)}$ is the compounded state transition (system Jacobian) matrix from timestep 0 to k. For a given block row of this matrix, we have:

$$\mathbf{H}_k\mathbf{\Phi}_{(k,0)} = \mathbf{H}_{proj,k}\, {}^{I_k}_{G}\mathbf{R}\begin{bmatrix}\mathbf{\Gamma}_1 & \mathbf{\Gamma}_2 & \mathbf{\Gamma}_3 & \mathbf{\Gamma}_4\end{bmatrix}$$

$$\begin{aligned} \mathbf{\Gamma}_1 &= \left\lfloor \left({}^G\mathbf{p}_f - {}^G\mathbf{p}_{I_k} + {}^{I_0}_{G}\mathbf{R}^\top\boldsymbol{\alpha}(k,0)\right)\times\right\rfloor {}^{I_0}_{G}\mathbf{R}^\top \\ &= \left\lfloor \left({}^G\mathbf{p}_f - {}^G\mathbf{p}_{I_0} - {}^G\mathbf{v}_{I_0}(t_k - t_0) - \frac{1}{2}{}^G\mathbf{g}(t_k-t_0)^2\right)\times\right\rfloor {}^{I_0}_{G}\mathbf{R}^\top \\ \mathbf{\Gamma}_2 &= -\mathbf{I}_{3\times3} \\ \mathbf{\Gamma}_3 &= -(t_k - t_0)\mathbf{I}_{3\times3} \\ \mathbf{\Gamma}_4 &= \mathbf{I}_{3\times3} \end{aligned}$$

We now verify the following nullspace which corresponds to the global yaw about gravity and global IMU and feature positions:

$$\mathcal{N}_{vins} = \begin{bmatrix} {}^{I_0}_{G}\mathbf{R}^G\mathbf{g} & \mathbf{0}_{3\times3} \\ -\lfloor {}^G\mathbf{p}_{I_0}\times\rfloor {}^G\mathbf{g} & \mathbf{I}_{3\times3} \\ -\lfloor {}^G\mathbf{v}_{I_0}\times\rfloor {}^G\mathbf{g} & \mathbf{0}_{3\times3} \\ -\lfloor {}^G\mathbf{p}_{f}\times\rfloor {}^G\mathbf{g} & \mathbf{I}_{3\times3} \end{bmatrix}$$

It is not difficult to verify that $\mathbf{H}_k\mathbf{\Phi}_{(k,0)}\mathcal{N}_{vio} = \mathbf{0}$. Thus this is a nullspace of the system, which clearly shows that there are the four unobserable directions (global yaw and position) of visual-inertial systems.

4.3 First Estimate Jacobians

The main idea of First-Estimate Jacobains (FEJ) approaches is to ensure that the state transition and Jacobian matrices are evaluated at correct linearization points such that the above observability analysis will hold true. For those interested in the technical details please take a look at: Huang et al. [2010] and Li and Mourikis [2013]. Let us first consider a small thought experiment of how the standard Kalman filter computes its state transition matrix. From a timestep zero to one it will use the current estimates from state zero forward in time. At the next timestep after it updates the state with measurements from other sensors, it will compute the state transition with the updated values to evolve the state to timestep two. This causes a miss-match in the "continuity" of the state transition matrix which when multiply sequentially should represent the evolution from time zero to time two.

$$\mathbf{\Phi}_{(k+1,k-1)}(\mathbf{x}_{k+1|k},\mathbf{x}_{k-1|k-1}) \neq \mathbf{\Phi}_{(k+1,k)}(\mathbf{x}_{k+1|k},\mathbf{x}_{k|k})\ \mathbf{\Phi}_{(k,k-1)}(\mathbf{x}_{k|k-1},\mathbf{x}_{k-1|k-1})$$

As shown above, we wish to compute the state transition matrix from the *k-1* timestep given all *k-1* measurements up until the current propagated timestep *k+1* given all *k* measurements. The right side of the above equation is how one would normally perform this in a Kalman filter framework. $\mathbf{\Phi}_{(k,k-1)}(\mathbf{x}_{k|k-1},\mathbf{x}_{k-1|k-1})$ corresponds to propagating from the *k-1* update time to the *k* timestep. One would then normally perform the *k*'th update to the state and then propagate from this **updated** state to the newest timestep (i.e. the $\mathbf{\Phi}_{(k+1,k)}(\mathbf{x}_{k+1|k},\mathbf{x}_{k|k})$ state transition matrix). This clearly is different then if one was to compute the state transition from time *k-1* to the *k+1* timestep as the second state transition is evaluated at the different $\mathbf{x}_{k|k}$ linearization point! To fix this, we can change the linearization point we evaluate these at:

$$\mathbf{\Phi}_{(k+1,k-1)}(\mathbf{x}_{k+1|k},\mathbf{x}_{k-1|k-1}) = \mathbf{\Phi}_{(k+1,k)}(\mathbf{x}_{k+1|k},\mathbf{x}_{k|k-1})\ \mathbf{\Phi}_{(k,k-1)}(\mathbf{x}_{k|k-1},\mathbf{x}_{k-1|k-1})$$

We also need to ensure that our measurement Jacobians match the linearization point of the state transition matrix. Thus they also need to be evaluated at the $\mathbf{x}_{k|k-1}$ linearization point instead of the $\mathbf{x}_{k|k}$ that one would normally use. This gives way to the name FEJ since we will evaluate the Jacobians at the same linearization point to ensure that the nullspace remains valid. For example if we evaluated the $\mathbf{H}_k$ Jacobian with a different ${}^G\mathbf{p}_f$ at each timestep then the nullspace would not hold past the first time instance.

Chapter 5

Measurement Update Derivations

5.1 Minimum Mean Square Error (MMSE) Estimation

Consider the following static state estimation problem: Given a prior distribution (probability density function or pdf) for a Gaussian random vector $\mathbf{x} \sim \mathcal{N}(\hat{\mathbf{x}}^{\ominus}, \mathbf{P}_{xx}^{\ominus})$ with dimension of n and a new m dimentional measurement $\mathbf{z}_m = \mathbf{z} + \mathbf{n} = \mathbf{h}(\mathbf{x}) + \mathbf{n}$ corrupted by zero-mean white Gaussian noise independent of state, $\mathbf{n} \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$, we want to compute the first two (central) moments of the posterior pdf $p(\mathbf{x}|\mathbf{z}_m)$. Generally (given a nonlinear measurement model), we approximate the posterior pdf as: $p(\mathbf{x}|\mathbf{z}_m) \simeq \mathcal{N}(\hat{\mathbf{x}}^{\oplus}, \mathbf{P}_{xx}^{\oplus})$. By design, this is the (approximate) solution to the MMSE estimation problem [Kay 1993] Kay [1993].

5.2 Conditional Probability Distribution

To this end, we employ the Bayes Rule:

$$p(\mathbf{x}|\mathbf{z}_m) = \frac{p(\mathbf{x}, \mathbf{z}_m)}{p(\mathbf{z}_m)}$$

In general, this conditional pdf cannot be computed analytically without imposing simplifying assumptions. For the problem at hand, we first approximate (if indeed) $p(\mathbf{z}_m) \simeq \mathcal{N}(\hat{\mathbf{z}}, \mathbf{P}_{zz})$, and then have the following joint Gaussian pdf (noting that joint of Gaussian pdfs is Gaussian):

$$p(\mathbf{x}, \mathbf{z}_m) = \mathcal{N}\left(\begin{bmatrix}\hat{\mathbf{x}}^{\ominus} \\ \mathbf{z}\end{bmatrix}, \begin{bmatrix}\mathbf{P}_{xx} & \mathbf{P}_{xz} \\ \mathbf{P}_{zx} & \mathbf{P}_{zz}\end{bmatrix}\right) =: \mathcal{N}(\hat{\mathbf{y}}, \mathbf{P}_{yy})$$

Substitution of these two Gaussians into the first equation yields the following conditional Gaussian pdf:

$$
\begin{aligned}
p(\mathbf{x}|\mathbf{z}_m) &\simeq \frac{\mathcal{N}(\hat{\mathbf{y}},\mathbf{P}_{yy})}{\mathcal{N}(\hat{\mathbf{z}},\mathbf{P}_{zz})} \\
&= \frac{\frac{1}{\sqrt{(2\pi)^{n+m}|\mathbf{P}_{yy}|}}e^{-\frac{1}{2}(\mathbf{y}-\hat{\mathbf{y}})^\top\mathbf{P}_{yy}^{-1}(\mathbf{y}-\hat{\mathbf{y}})}}{\frac{1}{\sqrt{(2\pi)^{m}|\mathbf{P}_{zz}|}}e^{-\frac{1}{2}(\mathbf{z}_m-\hat{\mathbf{z}})^\top\mathbf{P}_{zz}^{-1}(\mathbf{z}_m-\hat{\mathbf{z}})}} \\
&= \frac{1}{\sqrt{(2\pi)^{n}|\mathbf{P}_{yy}|/|\mathbf{P}_{zz}|}}e^{-\frac{1}{2}\left[(\mathbf{y}-\hat{\mathbf{y}})^\top\mathbf{P}_{yy}^{-1}(\mathbf{y}-\hat{\mathbf{y}})-(\mathbf{z}_m-\hat{\mathbf{z}})^\top\mathbf{P}_{zz}^{-1}(\mathbf{z}_m-\hat{\mathbf{z}})\right]} \\
&=: \mathcal{N}(\hat{\mathbf{x}}^\oplus,\mathbf{P}_{xx}^\oplus)
\end{aligned}
$$

We now derive the conditional mean and covariance can be computed as follows: First we simplify the denominator term $|\mathbf{P}_{yy}|/|\mathbf{P}_{zz}|$ in order to find the conditional covariance.

$$
|\mathbf{P}_{yy}| = \left|\begin{bmatrix}\mathbf{P}_{xx} & \mathbf{P}_{xz}\\ \mathbf{P}_{zx} & \mathbf{P}_{zz}\end{bmatrix}\right| = \left|\mathbf{P}_{xx}-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\right|\left|\mathbf{P}_{zz}\right|
$$

where we assumed $\mathbf{P}_{zz}$ is invertible and employed the determinant property of Schur complement. Thus, we have:

$$
\frac{|\mathbf{P}_{yy}|}{|\mathbf{P}_{zz}|} = \frac{\left|\mathbf{P}_{xx}-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\right|\left|\mathbf{P}_{zz}\right|}{|\mathbf{P}_{zz}|} = \left|\mathbf{P}_{xx}-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\right|
$$

Next, by defining the error states $\mathbf{r}_x = \mathbf{x}-\hat{\mathbf{x}}^\ominus$, $\mathbf{r}_z = \mathbf{z}_m-\hat{\mathbf{z}}$, $\mathbf{r}_y = \mathbf{y}-\hat{\mathbf{y}}$, and using the matrix inersion lemma, we rewrite the exponential term as follows:

$$
\begin{aligned}
&(\mathbf{y}-\hat{\mathbf{y}})^\top\mathbf{P}_{yy}^{-1}(\mathbf{y}-\hat{\mathbf{y}})-(\mathbf{z}_m-\hat{\mathbf{z}})^\top\mathbf{P}_{zz}^{-1}(\mathbf{z}_m-\hat{\mathbf{z}}) \\
&= \mathbf{r}_y^\top\mathbf{P}_{yy}^{-1}\mathbf{r}_y - \mathbf{r}_z^\top\mathbf{P}_{zz}^{-1}\mathbf{r}_z \\
&= \begin{bmatrix}\mathbf{r}_x\\ \mathbf{r}_z\end{bmatrix}^\top \begin{bmatrix}\mathbf{P}_{xx} & \mathbf{P}_{xz}\\ \mathbf{P}_{zx} & \mathbf{P}_{zz}\end{bmatrix}^{-1}\begin{bmatrix}\mathbf{r}_x\\ \mathbf{r}_z\end{bmatrix} - \mathbf{r}_z^\top\mathbf{P}_{zz}^{-1}\mathbf{r}_z \\
&= \begin{bmatrix}\mathbf{r}_x\\ \mathbf{r}_z\end{bmatrix}^\top \begin{bmatrix}\mathbf{Q} & -\mathbf{Q}\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\\ -\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\mathbf{Q} & \mathbf{P}_{zz}^{-1}+\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\mathbf{Q}\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\end{bmatrix}\begin{bmatrix}\mathbf{r}_x\\ \mathbf{r}_z\end{bmatrix} - \mathbf{r}_z^\top\mathbf{P}_{zz}^{-1}\mathbf{r}_z \\
&\qquad\qquad\text{where } \mathbf{Q} = (\mathbf{P}_{xx}-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx})^{-1} \\
&= \mathbf{r}_x^\top\mathbf{Q}\mathbf{r}_x - \mathbf{r}_x^\top\mathbf{Q}\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z - \mathbf{r}_z^\top\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\mathbf{Q}\mathbf{r}_x \\
&\qquad + \mathbf{r}_z^\top(\mathbf{P}_{zz}^{-1}+\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}\mathbf{Q}\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1})\mathbf{r}_z - \mathbf{r}_z^\top\mathbf{P}_{zz}^{-1}\mathbf{r}_z \\
&= \mathbf{r}_x^\top\mathbf{Q}\mathbf{r}_x - \mathbf{r}_x^\top\mathbf{Q}[\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_x] - [\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z]^\top\mathbf{Q}\mathbf{r}_x + [\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z]^\top\mathbf{Q}[\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z] \\
&= (\mathbf{r}_x-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z)^\top\mathbf{Q}(\mathbf{r}_x-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z) \\
&= (\mathbf{r}_x-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z)^\top(\mathbf{P}_{xx}-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx})^{-1}(\mathbf{r}_x-\mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z)
\end{aligned}
$$

where $(\mathbf{P}_{zz}^{-1})^\top = \mathbf{P}_{zz}^{-1}$ since covariance matrices are symmetric. Up to this point, we can now construct the conditional Gaussian pdf as follows:

$$\begin{aligned} p(\mathbf{x}_k|\mathbf{z}_m) = & \frac{1}{\sqrt{(2\pi)^n|\mathbf{P}_{xx} - \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx}|}} \times \\ & \exp\left(-\frac{1}{2}\left[(\mathbf{r}_x - \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z)^\top(\mathbf{P}_{xx} - \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx})^{-1}(\mathbf{r}_x - \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{r}_z)\right]\right) \end{aligned}$$

which results in the following conditional mean and covariance we were seeking:

$$\begin{aligned} \hat{\mathbf{x}}^\oplus &= \hat{\mathbf{x}}^\ominus + \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}(\mathbf{z}_m - \hat{\mathbf{z}}) \\ \mathbf{P}_{xx}^\oplus &= \mathbf{P}_{xx}^\ominus - \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}\mathbf{P}_{zx} \end{aligned}$$

These are the fundamental equations for (linear) state estimation.

5.3 Linear Measurement Update

As a special case, we consider a simple linear measurement model to illustrate the linear MMSE estimator:

$$\begin{aligned} \mathbf{z}_{m,k} &= \mathbf{H}_k\mathbf{x}_k + \mathbf{n}_k \\ \hat{\mathbf{z}}_k &:= \mathbb{E}[\mathbf{z}_{m,k}] = \mathbb{E}[\mathbf{H}_k\mathbf{x}_k + \mathbf{n}_k] = \mathbf{H}_k\hat{\mathbf{x}}_k^\ominus \end{aligned}$$

With this, we can derive the covariance and cross-correlation matrices as follows:

$$\begin{aligned} \mathbf{P}_{zz} &= \mathbb{E}\left[(\mathbf{z}_{m,k} - \hat{\mathbf{z}}_k)(\mathbf{z}_{m,k} - \hat{\mathbf{z}}_k)^\top\right] \\ &= \mathbb{E}\left[(\mathbf{H}_k\mathbf{x}_k + \mathbf{n}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^\ominus)(\mathbf{H}_k\mathbf{x}_k + \mathbf{n}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^\ominus)^\top\right] \\ &= \mathbb{E}\left[(\mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus) + \mathbf{n}_k)(\mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus) + \mathbf{n}_k)^\top\right] \\ &= \mathbb{E}\Big[\mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)^\top\mathbf{H}_k^\top + \mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)\mathbf{n}_k^\top \\ &\qquad + \mathbf{n}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)^\top\mathbf{H}_k^\top + \mathbf{n}_k\mathbf{n}_k^\top\Big] \\ &= \mathbb{E}\left[\mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)^\top\mathbf{H}_k^\top + \mathbf{n}_k\mathbf{n}_k^\top\right] \\ &= \mathbf{H}_k\,\mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)(\mathbf{x}_k - \hat{\mathbf{x}}_k^\ominus)^\top\right]\mathbf{H}_k^\top + \mathbb{E}\left[\mathbf{n}_k\mathbf{n}_k^\top\right] \\ &= \mathbf{H}_k\mathbf{P}_{xx}^\ominus\mathbf{H}_k^\top + \mathbf{R}_k \end{aligned}$$

where $\mathbf{R}_k$ is the *discrete* measurement noise matrix, $\mathbf{H}_k$ is the measurement Jacobian mapping the state into the measurement domain, and $\mathbf{P}_{xx}^\ominus$ is the current state covariance.

$$
\begin{aligned}
\mathbf{P}_{xz} &= \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})(\mathbf{z}_{m,k} - \hat{\mathbf{z}}_k)^\top\right] \\
&= \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})(\mathbf{H}_k\mathbf{x}_k + \mathbf{n}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^{\ominus})^\top\right] \\
&= \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})(\mathbf{H}_k(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus}) + \mathbf{n}_k)^\top\right] \\
&= \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})^\top\mathbf{H}_k^\top + (\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})\mathbf{n}_k^\top\right] \\
&= \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})^\top\right]\mathbf{H}_k^\top + \mathbb{E}\left[(\mathbf{x}_k - \hat{\mathbf{x}}_k^{\ominus})\mathbf{n}_k^\top\right] \\
&= \mathbf{P}_{xx}^{\ominus}\mathbf{H}_k^\top
\end{aligned}
$$

where we have employed the fact that the noise is independent of the state. Substitution of these quantities into the fundamental equation leads to the following update equations:

$$
\begin{aligned}
\hat{\mathbf{x}}_k^{\oplus} &= \hat{\mathbf{x}}_k^{\ominus} + \mathbf{P}_k^{\ominus}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_k^{\ominus}\mathbf{H}_k^\top + \mathbf{R}_k)^{-1}(\mathbf{z}_{m,k} - \hat{\mathbf{z}}_k) \\
&= \hat{\mathbf{x}}_k^{\ominus} + \mathbf{P}_k^{\ominus}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_k^{\ominus}\mathbf{H}_k^\top + \mathbf{R}_k)^{-1}(\mathbf{z}_{m,k} - \mathbf{H}_k\hat{\mathbf{x}}_k^{\ominus}) \\
&= \hat{\mathbf{x}}_k^{\ominus} + \mathbf{K}\mathbf{r}_z \\
\mathbf{P}_{xx}^{\oplus} &= \mathbf{P}_k^{\ominus} - \mathbf{P}_k^{\ominus}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_k^{\ominus}\mathbf{H}_k^\top + \mathbf{R}_k)^{-1}(\mathbf{P}_k^{\ominus}\mathbf{H}_k^\top)^\top \\
&= \mathbf{P}_k^{\ominus} - \mathbf{P}_k^{\ominus}\mathbf{H}_k^\top(\mathbf{H}_k\mathbf{P}_k^{\ominus}\mathbf{H}_k^\top + \mathbf{R}_k)^{-1}\mathbf{H}_k\mathbf{P}_k^{\ominus}
\end{aligned}
$$

These are essentially the Kalman filter (or linear MMSE) update equations.

5.4 Update Equations and Derivations

- Feature Triangulation — 3D feature triangulation derivations for getting a feature linearization point
- Camera Measurement Update — Measurement equations and derivation for 3D feature point
- Delayed Feature Initialization — How to perform delayed initialization
- MSCKF Nullspace Projection — MSCKF nullspace projection
- Measurement Compression — MSCKF measurement compression
- Zero Velocity Update — Zero velocity stationary update

5.5 Feature Triangulation

5.5.1 3D Cartesian Triangulation

We wish to create a solvable linear system that can give us an initial guess for the 3D cartesian position of our feature. To do this, we take all the poses that the feature is seen from to be of known quantity. This feature will be triangulated in some anchor camera frame $\{A\}$ which we can arbitrary pick. If the feature $\mathbf{p}_f$ is observed by pose $1 \ldots m$, given the anchor pose A, we can have the following transformation from any camera pose $C_i, i = 1 \ldots m$:

$$\begin{aligned} {}^{C_i}\mathbf{p}_f &= {}^{C_i}_{A}\mathbf{R}\left({}^{A}\mathbf{p}_f - {}^{A}\mathbf{p}_{C_i}\right) \\ {}^{A}\mathbf{p}_f &= {}^{C_i}_{A}\mathbf{R}^{\top}\,{}^{C_i}\mathbf{p}_f + {}^{A}\mathbf{p}_{C_i} \end{aligned}$$

In the absents of noise, the measurement in the current frame is the bearing ${}^{C_i}\mathbf{b}$ and its depth ${}^{C_i}z$. Thus we have the following mapping to a feature seen from the current frame:

$${}^{C_i}\mathbf{p}_f = {}^{C_i}z_f\,{}^{C_i}\mathbf{b}_f = {}^{C_i}z_f \begin{bmatrix} u_n \\ v_n \\ 1 \end{bmatrix}$$

We note that u_n and v_n represent the undistorted normalized image coordinates. This bearing can be warped into the the anchor frame by substituting into the above equation:

$$\begin{aligned} {}^{A}\mathbf{p}_f &= {}^{C_i}_{A}\mathbf{R}^{\top}\,{}^{C_i}z_f\,{}^{C_i}\mathbf{b}_f + {}^{A}\mathbf{p}_{C_i} \\ &= {}^{C_i}z_f\,{}^{A}\mathbf{b}_{C_i\to f} + {}^{A}\mathbf{p}_{C_i} \end{aligned}$$

To remove the need to estimate the extra degree of freedom of depth ${}^{C_i}z_f$, we define the following vectors which are orthoganal to the bearing ${}^{A}\mathbf{b}_{C_i\to f}$:

$${}^{A}\mathbf{N}_i = \lfloor {}^{A}\mathbf{b}_{C_i\to f}\times\rfloor = \begin{bmatrix} 0 & -{}^{A}b_{C_i\to f}(3) & {}^{A}b_{C_i\to f}(2) \\ {}^{A}b_{C_i\to f}(3) & 0 & -{}^{A}b_{C_i\to f}(1) \\ -{}^{A}b_{C_i\to f}(2) & {}^{A}b_{C_i\to f}(1) & 0 \end{bmatrix}$$

All three rows are perpendicular to the vector ${}^{A}\mathbf{b}_{C_i\to f}$ and thus ${}^{A}\mathbf{N}_i\,{}^{A}\mathbf{b}_{C_i\to f} = \mathbf{0}_3$. We can then multiple the transform equation/constraint to form two equation which only relates to the unknown 3 d.o.f ${}^{A}\mathbf{p}_f$:

$$\begin{aligned} {}^{A}\mathbf{N}_i\,{}^{A}\mathbf{p}_f &= {}^{A}\mathbf{N}_i\,{}^{C_i}z_f\,{}^{A}\mathbf{b}_{C_i\to f} + {}^{A}\mathbf{N}_i\,{}^{A}\mathbf{p}_{C_i} \\ &= {}^{A}\mathbf{N}_i\,{}^{A}\mathbf{p}_{C_i} \end{aligned}$$

By stacking all the measurements, we can have:

$$\underbrace{\begin{bmatrix} \vdots \\ {}^{A}\mathbf{N}_i \\ \vdots \end{bmatrix}}_{\mathbf{A}} {}^{A}\mathbf{p}_f = \underbrace{\begin{bmatrix} \vdots \\ {}^{A}\mathbf{N}_i\,{}^{A}\mathbf{p}_{C_i} \\ \vdots \end{bmatrix}}_{\mathbf{b}}$$

Since each pixel measurement provides two constraints, as long as $m > 1$, we will have enough constraints to triangulate the feature. In practice, the more views of the feature the better the triangulation and thus normally want to have a feature seen from at least five views. We could select two rows of the each ${}^{A}\mathbf{N}_i$ to reduce the number of rows, but by having a square system we can perform the following "trick".

$$\begin{aligned}\mathbf{A}^\top\mathbf{A}\ {}^{A}\mathbf{p}_f &= \mathbf{A}^\top\mathbf{b} \\ \Big(\sum_i {}^{A}\mathbf{N}_i^\top {}^{A}\mathbf{N}_i\Big){}^{A}\mathbf{p}_f &= \Big(\sum_i {}^{A}\mathbf{N}_i^\top {}^{A}\mathbf{N}_i {}^{A}\mathbf{p}_{C_i}\Big)\end{aligned}$$

This is a 3x3 system which can be quickly solved for as compared to the originl 3mx3m or 2mx2m system. We additionally check that the triangulated feature is "valid" and in front of the camera and not too far away. The condition number of the above linear system and reject systems that are "sensitive" to errors and have a large value.

5.5.2 1D Depth Triangulation

We wish to create a solvable linear system that can give us an initial guess for the 1D depth position of our feature. To do this, we take all the poses that the feature is seen from to be of known quantity along with the bearing in the anchor frame. This feature will be triangulated in some anchor camera frame $\{A\}$ which we can arbitrary pick. We define it as its normalized image coordiantes $[u_n\ v_n\ 1]^\top$ in tha anchor frame. If the feature $\mathbf{p}_f$ is observed by pose $1\ldots m$, given the anchor pose A, we can have the following transformation from any camera pose $C_i, i = 1\ldots m$:

$$\begin{aligned}{}^{C_i}\mathbf{p}_f &= {}^{C_i}_{A}\mathbf{R}\left({}^{A}\mathbf{p}_f - {}^{A}\mathbf{p}_{C_i}\right) \\ {}^{A}\mathbf{p}_f &= {}^{C_i}_{A}\mathbf{R}^\top {}^{C_i}\mathbf{p}_f + {}^{A}\mathbf{p}_{C_i} \\ {}^{A}z_f {}^{A}\mathbf{b}_f &= {}^{C_i}_{A}\mathbf{R}^\top {}^{C_i}\mathbf{p}_f + {}^{A}\mathbf{p}_{C_i}\end{aligned}$$

In the absents of noise, the measurement in the current frame is the bearing ${}^{C_i}\mathbf{b}$ and its depth ${}^{C_i}z$.

$${}^{C_i}\mathbf{p}_f = {}^{C_i}z_f {}^{C_i}\mathbf{b}_f = {}^{C_i}z_f \begin{bmatrix} u_n \\ v_n \\ 1 \end{bmatrix}$$

We note that u_n and v_n represent the undistorted normalized image coordinates. This bearing can be warped into the the anchor frame by substituting into the above equation:

$$\begin{aligned}{}^{A}z_f {}^{A}\mathbf{b}_f &= {}^{C_i}_{A}\mathbf{R}^\top {}^{C_i}z_f {}^{C_i}\mathbf{b}_f + {}^{A}\mathbf{p}_{C_i} \\ &= {}^{C_i}z_f {}^{A}\mathbf{b}_{C_i\rightarrow f} + {}^{A}\mathbf{p}_{C_i}\end{aligned}$$

To remove the need to estimate the extra degree of freedom of depth ${}^{C_i}z_f$, we define the following vectors which are orthoganal to the bearing ${}^{A}\mathbf{b}_{C_i\rightarrow f}$:

$${}^{A}\mathbf{N}_i = \lfloor {}^{A}\mathbf{b}_{C_i\rightarrow f} \times \rfloor$$

All three rows are perpendicular to the vector ${}^A\mathbf{b}_{C_i\to f}$ and thus ${}^A\mathbf{N}_i{}^A\mathbf{b}_{C_i\to f} = \mathbf{0}_3$. We can then multiple the transform equation/constraint to form two equation which only relates to the unknown ${}^Az_f$:

$$\begin{aligned}({}^A\mathbf{N}_i{}^A\mathbf{b}_f){}^Az_f &= {}^A\mathbf{N}_i{}^{C_i}z_f{}^A\mathbf{b}_{C_i\to f} + {}^A\mathbf{N}_i{}^A\mathbf{p}_{C_i}\\ &= {}^A\mathbf{N}_i{}^A\mathbf{p}_{C_i}\end{aligned}$$

We can then formulate the following system:

$$\Big(\sum_i({}^A\mathbf{N}_i{}^A\mathbf{b}_f)^\top({}^A\mathbf{N}_i{}^A\mathbf{b}_f)\Big){}^Az_f = \Big(\sum_i({}^A\mathbf{N}_i{}^A\mathbf{b}_f)^\top{}^A\mathbf{N}_i{}^A\mathbf{b}_i\Big)$$

This is a 1x1 system which can be quickly solved for with a single scalar division. We additionally check that the triangulated feature is "valid" and in front of the camera and not too far away. The full feature can be reconstructed by ${}^A\mathbf{p}_f = {}^Az_f{}^A\mathbf{b}_f$.

5.5.3 3D Inverse Non-linear Optimization

After we get the triangulated feature 3D position, a nonlinear least-squares will be performed to refine this estimate. In order to achieve good numerical stability, we use the inverse depth representation for point feature which helps with convergence. We find that in most cases this problem converges within 2-3 iterations in indoor environments. The feature transformation can be written as:

$$\begin{aligned}{}^{C_i}\mathbf{p}_f &= {}^{C_i}_A\mathbf{R}\left({}^A\mathbf{p}_f - {}^A\mathbf{p}_{C_i}\right)\\ &= {}^Az_f{}^{C_i}_A\mathbf{R}\left(\begin{bmatrix}{}^Ax_f/{}^Az_f\\ {}^Ay_f/{}^Az_f\\ 1\end{bmatrix} - \frac{1}{{}^Az_f}{}^A\mathbf{p}_{C_i}\right)\\ \Rightarrow \frac{1}{{}^Az_f}{}^{C_i}\mathbf{p}_f &= {}^{C_i}_A\mathbf{R}\left(\begin{bmatrix}{}^Ax_f/{}^Az_f\\ {}^Ay_f/{}^Az_f\\ 1\end{bmatrix} - \frac{1}{{}^Az_f}{}^A\mathbf{p}_{C_i}\right)\end{aligned}$$

We define $u_A = {}^Ax_f/{}^Az_f$, $v_A = {}^Ay_f/{}^Az_f$, and $\rho_A = 1/{}^Az_f$ to get the following measurement equation:

$$h(u_A, v_A, \rho_A) = {}^{C_i}_A\mathbf{R}\left(\begin{bmatrix}u_A\\ v_A\\ 1\end{bmatrix} - \rho_A{}^A\mathbf{p}_{C_i}\right)$$

The feature measurement seen from the $\{C_i\}$ camera frame can be reformulated as:

$$\begin{aligned}
\mathbf{z} &= \begin{bmatrix} u_i \\ v_i \end{bmatrix} \\
&= \begin{bmatrix} h(u_A, v_A, \rho_A)(1)/h(u_A, v_A, \rho_A)(3) \\ h(u_A, v_A, \rho_A)(2)/h(u_A, v_A, \rho_A)(3) \end{bmatrix} \\
&= \mathbf{h}(u_A, v_A, \rho_A)
\end{aligned}$$

Therefore, we can have the least-squares formulated and Jacobians:

$$\underset{u_A, v_A, \rho_A}{\operatorname{argmin}} ||\mathbf{z} - \mathbf{h}(u_A, v_A, \rho_A)||^2$$

$$\begin{aligned}
\frac{\partial \mathbf{h}(u_A, v_A, \rho_A)}{\partial h(u_A, v_A, \rho_A)} &= \begin{bmatrix} 1/h(\cdots)(1) & 0 & -h(\cdots)(1)/h(\cdots)(3)^2 \\ 0 & 1/h(\cdots)(2) & -h(\cdots)(2)/h(\cdots)(3)^2 \end{bmatrix} \\
\frac{\partial h(u_A, v_A, \rho_A)}{\partial [u_A, v_A, \rho_A]} &= {}^{C_i}_{A}\mathbf{R} \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} & -{}^{A}\mathbf{p}_{C_i} \end{bmatrix}
\end{aligned}$$

The least-squares problem can be solved with `Gaussian-Newton` or `Levenberg-Marquart` algorithm.

5.6 Camera Measurement Update

5.6.1 Perspective Projection (Bearing) Measurement Model

Consider a 3D feature is detected from the camera image at time k, whose uv measurement (i.e., the corresponding pixel coordinates) on the image plane is given by:

$$\begin{aligned}
\mathbf{z}_{m,k} &= \mathbf{h}(\mathbf{x}_k) + \mathbf{n}_k \\
&= \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta}) + \mathbf{n}_k \\
&= \mathbf{h}_d(\mathbf{h}_p({}^{C_k}\mathbf{p}_f),\ \boldsymbol{\zeta}) + \mathbf{n}_k \\
&= \mathbf{h}_d(\mathbf{h}_p(\mathbf{h}_t({}^{G}\mathbf{p}_f,\ {}^{C_k}_{G}\mathbf{R},\ {}^{G}\mathbf{p}_{C_k})),\ \boldsymbol{\zeta}) + \mathbf{n}_k \\
&= \mathbf{h}_d(\mathbf{h}_p(\mathbf{h}_t(\mathbf{h}_r(\boldsymbol{\lambda}, \cdots),\ {}^{C_k}_{G}\mathbf{R},\ {}^{G}\mathbf{p}_{C_k})),\ \boldsymbol{\zeta}) + \mathbf{n}_k
\end{aligned}$$

where $\mathbf{n}_k$ is the measurement noise and typically assumed to be zero-mean white Gaussian; $\mathbf{z}_{n,k}$ is the normalized undistorted uv measurement; $\boldsymbol{\zeta}$ is the camera intrinsic parameters such as focal length and distortion parameters; ${}^{C_k}\mathbf{p}_f$ is the feature position in the current camera frame $\{C_k\}$; ${}^{G}\mathbf{p}_f$ is the feature position in the global frame $\{G\}$; $\{{}^{C_k}_{G}\mathbf{R},\ {}^{G}\mathbf{p}_{C_k}\}$ denotes the current camera pose (position and orientation) in the global frame (or camera extrinsics); and $\boldsymbol{\lambda}$ is the feature's parameters of different representations (other than position) such as simply a xyz position or an inverse depth with bearing.

In the above expression, we decompose the measurement function into multiple concatenated functions corresponding to different operations, which map the states into the raw uv measurement on the image plane. It should be noted that as we will perform intrinsic calibration along with extrinsic with different feature representations, the above camera measurement model is general. The high-level description of each function is given in the next section.

5.6.1.1 Measurement Function Overview

Function	Description
$\mathbf{z}_k = \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta})$	The distortion function that takes normalized coordinates and maps it into distorted uv coordinates
$\mathbf{z}_{n,k} = \mathbf{h}_p({}^{C_k}\mathbf{p}_f)$	The projection function that takes a 3D point in the image and converts it into the normalized uv coordinates
${}^{C_k}\mathbf{p}_f = \mathbf{h}_t({}^{G}\mathbf{p}_f,\ {}^{C_k}_{G}\mathbf{R},\ {}^{G}\mathbf{p}_{C_k})$	Transforming a feature's position in the global frame into the current camera frame
${}^{G}\mathbf{p}_f = \mathbf{h}_r(\boldsymbol{\lambda},\cdots)$	Converting from a feature representation to a 3D feature in the global frame

5.6.1.2 Jacobian Computation

Given the above nested functions, we can leverage the chainrule to find the total state Jacobian. Since our feature representation function $\mathbf{h}_r(\cdots)$ might also depend on the state, i.e. an anchoring pose, we need to carefully consider its additional derivatives. Consider the following example of our measurement in respect to a state $\mathbf{x}$ Jacobian:

$$\frac{\partial \mathbf{z}_k}{\partial \mathbf{x}} = \frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} \frac{\partial \mathbf{h}_p(\cdot)}{\partial {}^{C_k}\mathbf{p}_f} \frac{\partial \mathbf{h}_t(\cdot)}{\partial \mathbf{x}} + \frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} \frac{\partial \mathbf{h}_p(\cdot)}{\partial {}^{C_k}\mathbf{p}_f} \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^{G}\mathbf{p}_f} \frac{\partial \mathbf{h}_r(\cdot)}{\partial \mathbf{x}}$$

In the global feature representations, see Point Feature Representations section, the second term will be zero while for the anchored representations it will need to be computed.

5.6.2 Distortion Function

5.6.2.1 Radial model

To calibrate camera intrinsics, we need to know how to map our normalized coordinates into the raw pixel coordinates on the image plane. We first employ the radial distortion as in `OpenCV model`:

$$\begin{bmatrix} u \\ v \end{bmatrix} := \mathbf{z}_k = \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta}) = \begin{bmatrix} f_x * x + c_x \\ f_y * y + c_y \end{bmatrix}$$

$$\begin{aligned} \text{where}\ \ x &= x_n(1 + k_1 r^2 + k_2 r^4) + 2p_1 x_n y_n + p_2(r^2 + 2x_n^2) \\ y &= y_n(1 + k_1 r^2 + k_2 r^4) + p_1(r^2 + 2y_n^2) + 2p_2 x_n y_n \\ r^2 &= x_n^2 + y_n^2 \end{aligned}$$

where $\mathbf{z}_{n,k} = [x_n\ y_n]^\top$ are the normalized coordinates of the 3D feature and u and v are the distorted image coordinates on the image plane. The following distortion and camera intrinsic (focal length and image center) parameters are involved in the above distortion model, which can be estimated online:

$$\boldsymbol{\zeta} = \begin{bmatrix} f_x & f_y & c_x & c_y & k_1 & k_2 & p_1 & p_2 \end{bmatrix}^\top$$

Note that we do not estimate the higher order (i.e., higher than fourth order) terms as in most offline calibration methods such as Kalibr. To estimate these intrinsic parameters (including the distortation parameters), the following Jacobian for these parameters is needed:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \boldsymbol{\zeta}} = \begin{bmatrix} x & 0 & 1 & 0 & f_x*(x_n r^2) & f_x*(x_n r^4) & f_x*(2x_n y_n) & f_x*(r^2+2x_n^2) \\ 0 & y & 0 & 1 & f_y*(y_n r^2) & f_y*(y_n r^4) & f_y*(r^2+2y_n^2) & f_y*(2x_n y_n) \end{bmatrix}$$

Similarly, the Jacobian with respect to the normalized coordinates can be obtained as follows:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} = \Big[\begin{matrix} f_x*((1+k_1r^2+k_2r^4)+(2k_1x_n^2+4k_2x_n^2(x_n^2+y_n^2))+2p_1y_n+(2p_2x_n+4p_2x_n)) & f_x*(2k_1x_ny_n+4 \\ f_y*(2k_1x_ny_n+4k_2x_ny_n(x_n^2+y_n^2)+2p_1x_n+2p_2y_n) & f_y*((1+k_1r^2+k_2r^4)+(2k_1y \end{matrix}$$

5.6.2.2 Fisheye model

As fisheye or wide-angle lenses are widely used in practice, we here provide mathematical derivations of such distortion model as in OpenCV fisheye.

$$\begin{bmatrix} u \\ v \end{bmatrix} := \mathbf{z}_k = \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta}) = \begin{bmatrix} f_x*x+c_x \\ f_y*y+c_y \end{bmatrix}$$

$$\begin{aligned} \text{where}\ \ x &= \frac{x_n}{r}*\theta_d \\ y &= \frac{y_n}{r}*\theta_d \\ \theta_d &= \theta(1+k_1\theta^2+k_2\theta^4+k_3\theta^6+k_4\theta^8) \\ r^2 &= x_n^2+y_n^2 \\ \theta &= atan(r) \end{aligned}$$

where $\mathbf{z}_{n,k} = [x_n\ y_n]^\top$ are the normalized coordinates of the 3D feature and u and v are the distorted image coordinates on the image plane. Clearly, the following distortion intrinsic parameters are used in the above model:

$$\boldsymbol{\zeta} = \begin{bmatrix} f_x & f_y & c_x & c_y & k_1 & k_2 & k_3 & k_4 \end{bmatrix}^\top$$

In analogy to the previous radial distortion case, the following Jacobian for these parameters is needed for intrinsic calibration:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \boldsymbol{\zeta}} = \begin{bmatrix} x_n & 0 & 1 & 0 & f_x*(\frac{x_n}{r}\theta^3) & f_x*(\frac{x_n}{r}\theta^5) & f_x*(\frac{x_n}{r}\theta^7) & f_x*(\frac{x_n}{r}\theta^9) \\ 0 & y_n & 0 & 1 & f_y*(\frac{y_n}{r}\theta^3) & f_y*(\frac{y_n}{r}\theta^5) & f_y*(\frac{y_n}{r}\theta^7) & f_y*(\frac{y_n}{r}\theta^9) \end{bmatrix}$$

Similarly, with the chain rule of differentiation, we can compute the following Jacobian with respect to the normalized coordinates:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} = \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial x_n y_n} + \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial r}\frac{\partial r}{\partial x_n y_n} + \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial \theta_d}\frac{\partial \theta_d}{\partial \theta}\frac{\partial \theta}{\partial r}\frac{\partial r}{\partial x_n y_n}$$

$$\begin{aligned}
\text{where}\quad \frac{\partial uv}{\partial xy} &= \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \\
\frac{\partial xy}{\partial x_n y_n} &= \begin{bmatrix} \theta_d/r & 0 \\ 0 & \theta_d/r \end{bmatrix} \\
\frac{\partial xy}{\partial r} &= \begin{bmatrix} -\frac{x_n}{r^2}\theta_d \\ -\frac{y_n}{r^2}\theta_d \end{bmatrix} \\
\frac{\partial r}{\partial x_n y_n} &= \begin{bmatrix} \frac{x_n}{r} & \frac{y_n}{r} \end{bmatrix} \\
\frac{\partial xy}{\partial \theta_d} &= \begin{bmatrix} \frac{x_n}{r} \\ \frac{y_n}{r} \end{bmatrix} \\
\frac{\partial \theta_d}{\partial \theta} &= \begin{bmatrix} 1 + 3k_1\theta^2 + 5k_2\theta^4 + 7k_3\theta^6 + 9k_4\theta^8 \end{bmatrix} \\
\frac{\partial \theta}{\partial r} &= \begin{bmatrix} \frac{1}{r^2+1} \end{bmatrix}
\end{aligned}$$

5.6.3 Perspective Projection Function

The standard pinhole camera model is used to project a 3D point in the *camera* frame into the normalized image plane (with unit depth):

$$\mathbf{z}_{n,k} = \mathbf{h}_p({}^{C_k}\mathbf{p}_f) = \begin{bmatrix} {}^Cx/{}^Cz \\ {}^Cy/{}^Cz \end{bmatrix}$$

$$\text{where}\quad {}^{C_k}\mathbf{p}_f = \begin{bmatrix} {}^Cx \\ {}^Cy \\ {}^Cz \end{bmatrix}$$

whose Jacobian matrix is computed as follows:

$$\frac{\partial \mathbf{h}_p(\cdot)}{\partial {}^{C_k}\mathbf{p}_f} = \begin{bmatrix} \frac{1}{{}^Cz} & 0 & \frac{-{}^Cx}{({}^Cz)^2} \\ 0 & \frac{1}{{}^Cz} & \frac{-{}^Cy}{({}^Cz)^2} \end{bmatrix}$$

5.6.4 Euclidean Transformation

We employ the 6DOF rigid-body Euclidean transformation to transform the 3D feature position in the global frame $\{G\}$ to the current camera frame $\{C_k\}$ based on the current global camera pose:

$$ {}^{C_k}\mathbf{p}_f = \mathbf{h}_t({}^G\mathbf{p}_f,\ {}^{C_k}_{G}\mathbf{R},\ {}^G\mathbf{p}_{C_k}) = {}^{C_k}_{G}\mathbf{R}({}^G\mathbf{p}_f - {}^G\mathbf{p}_{C_k}) $$

Note that in visual-inertial navigation systems, we often keep the IMU, instead of camera, state in the state vector. So, we need to further transform the above geometry using the time-invariant IMU-camera extrinsic parameters $\{{}^C_I\mathbf{R},\ {}^C\mathbf{p}_I\}$ as follows:

$$ \begin{aligned} {}^G\mathbf{p}_{C_k} &= {}^G\mathbf{p}_{I_k} + {}^G_I\mathbf{R}\,{}^I\mathbf{p}_{C_k} = {}^G\mathbf{p}_{I_k} + {}^G_I\mathbf{R}\,{}^I\mathbf{p}_{C} \\ {}^{C_k}_{G}\mathbf{R} &= {}^{C_k}_{I}\mathbf{R}\,{}^{I_k}_{G}\mathbf{R} = {}^{C}_{I}\mathbf{R}\,{}^{I_k}_{G}\mathbf{R} \end{aligned} $$

Substituting these quantities into the equation of ${}^{C_k}\mathbf{p}_f$ yields:

$$ {}^{C_k}\mathbf{p}_f = {}^C_I\mathbf{R}\,{}^{I_k}_{G}\mathbf{R}({}^G\mathbf{p}_f - {}^G\mathbf{p}_{I_k}) + {}^C\mathbf{p}_I $$

We now can compute the following Jacobian with respect to the pertinent states:

$$ \begin{aligned} \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^G\mathbf{p}_f} &= {}^C_I\mathbf{R}\,{}^{I_k}_{G}\mathbf{R} \\ \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^{I_k}_{G}\mathbf{R}} &= {}^C_I\mathbf{R}\left\lfloor {}^{I_k}_{G}\mathbf{R}({}^G\mathbf{p}_f - {}^G\mathbf{p}_{I_k})\times \right\rfloor \\ \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^G\mathbf{p}_{I_k}} &= -{}^C_I\mathbf{R}\,{}^{I_k}_{G}\mathbf{R} \end{aligned} $$

where $\lfloor \mathbf{a}\times \rfloor$ denotes the skew symmetric matrix of a vector $\mathbf{a}$ (see `Quaternion TR` Trawny and Roumeliotis [2005]). Note also that in above expression (as well as in ensuing derivations), there is a little abuse of notation; that is, the Jacobian with respect to the rotation matrix is not the direct differentiation with respect to the 3x3 rotation matrix, instead with respect to the corresponding 3x1 rotation angle vector. Moreover, if performing online extrinsic calibration, the Jacobian with respect to the IMU-camera extrinsics is needed:

$$ \begin{aligned} \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^C_I\mathbf{R}} &= \left\lfloor {}^C_I\mathbf{R}\,{}^{I_k}_{G}\mathbf{R}({}^G\mathbf{p}_f - {}^G\mathbf{p}_{I_k})\times \right\rfloor \\ \frac{\partial \mathbf{h}_t(\cdot)}{\partial {}^C\mathbf{p}_I} &= \mathbf{I}_{3\times 3} \end{aligned} $$

5.6.5 Point Feature Representations

There are two main parameterizations of a 3D point feature: 3D position (xyz) and inverse depth with bearing. Both of these can either be represented in the global frame or in an anchor frame of reference which adds a dependency on having an "anchor" pose where the feature is observed. To allow for a unified treatment of different feature parameterizations $\boldsymbol{\lambda}$ in our codebase, we derive in detail the generic function ${}^G\mathbf{p}_f = \mathbf{f}(\cdot)$ that maps different representations into global position.

5.6.5.1 Global XYZ

As the canonical parameterization, the global position of a 3D point feature is simply given by its xyz coordinates in the global frame of reference:

$$\begin{aligned} {}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda}) \\ &= \begin{bmatrix} {}^Gx \\ {}^Gy \\ {}^Gz \end{bmatrix} \\ \text{where} \quad \boldsymbol{\lambda} &= {}^G\mathbf{p}_f = \begin{bmatrix} {}^Gx & {}^Gy & {}^Gz \end{bmatrix}^\top \end{aligned}$$

It is clear that the Jacobian with respect to the feature parameters is:

$$\frac{\partial \mathbf{f}(\cdot)}{\partial \boldsymbol{\lambda}} = \mathbf{I}_{3\times 3}$$

5.6.5.2 Global Inverse Depth

The global inverse-depth representation of a 3D point feature is given by (akin to spherical coordinates):

$$\begin{aligned} {}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda}) \\ &= \frac{1}{\rho} \begin{bmatrix} \cos(\theta)\sin(\phi) \\ \sin(\theta)\sin(\phi) \\ \cos(\phi) \end{bmatrix} \\ \text{where} \quad \boldsymbol{\lambda} &= \begin{bmatrix} \theta & \phi & \rho \end{bmatrix}^\top \end{aligned}$$

The Jacobian with respect to the feature parameters can be computed as:

$$\frac{\partial \mathbf{f}(\cdot)}{\partial \boldsymbol{\lambda}} = \begin{bmatrix} -\frac{1}{\rho}\sin(\theta)\sin(\phi) & \frac{1}{\rho}\cos(\theta)\cos(\phi) & -\frac{1}{\rho^2}\cos(\theta)\sin(\phi) \\ \frac{1}{\rho}\cos(\theta)\sin(\phi) & \frac{1}{\rho}\sin(\theta)\cos(\phi) & -\frac{1}{\rho^2}\sin(\theta)\sin(\phi) \\ 0 & -\frac{1}{\rho}\sin(\phi) & -\frac{1}{\rho^2}\cos(\phi) \end{bmatrix}$$

5.6.5.3 Global Inverse Depth (MSCKF VERSION)

Note that as this representation has a singularity when the z-distance goes to zero, it is not recommended to use in practice. Instead, one should use the Anchored Inverse Depth (MSCKF Version) representation. The anchored version doesn't have this issue if features are represented in a camera frame that they where seen from (in which features should never have a non-positive z-direction).

5.6.5.4 Anchored XYZ

We can represent a 3D point feature in some "anchor" frame (say some IMU local frame, $\{^{I_a}_G\mathbf{R},\ ^G\mathbf{p}_{I_a}\}$), which would normally be the IMU pose corresponding to the first camera frame where the feature was detected.

$$\begin{aligned}
{}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda},\ ^{I_a}_G\mathbf{R},\ ^G\mathbf{p}_{I_a},\ ^C_I\mathbf{R},\ ^C\mathbf{p}_I) \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top(\boldsymbol{\lambda} - {}^C\mathbf{p}_I) + {}^G\mathbf{p}_{I_a} \\
\text{where}\quad \boldsymbol{\lambda} &= {}^{C_a}\mathbf{p}_f = \begin{bmatrix} {}^{C_a}x & {}^{C_a}y & {}^{C_a}z \end{bmatrix}^\top
\end{aligned}$$

The Jacobian with respect to the feature state is given by:

$$\frac{\partial \mathbf{f}(\cdot)}{\partial \boldsymbol{\lambda}} = {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top$$

As the anchor pose is involved in this representation, its Jacobians are computed as:

$$\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^{I_a}_G\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top \left\lfloor {}^C_I\mathbf{R}^\top({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I)\times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^G\mathbf{p}_{I_a}} &= \mathbf{I}_{3\times 3}
\end{aligned}$$

Moreover, if performing extrinsic calibration, the following Jacobians with respect to the IMU-camera extrinsics are also needed:

$$\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C_I\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left\lfloor ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I)\times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C\mathbf{p}_I} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top
\end{aligned}$$

5.6.5.5 Anchored Inverse Depth

In analogy to the global inverse depth case, we can employ the inverse-depth with bearing (akin to spherical coordinates) in the anchor frame, $\{^{I_a}_G\mathbf{R},\ ^G\mathbf{p}_{I_a}\}$, to represent a 3D point feature:

$$\begin{aligned}
{}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda},\ ^{I_a}_G\mathbf{R},\ ^G\mathbf{p}_{I_a},\ ^C_I\mathbf{R},\ ^C\mathbf{p}_I) \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) + {}^G\mathbf{p}_{I_a} \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left(\frac{1}{\rho}\begin{bmatrix}\cos(\theta)\sin(\phi)\\ \sin(\theta)\sin(\phi)\\ \cos(\phi)\end{bmatrix} - {}^C\mathbf{p}_I\right) + {}^G\mathbf{p}_{I_a} \\
\text{where}\quad \boldsymbol{\lambda} &= \begin{bmatrix}\theta & \phi & \rho\end{bmatrix}^\top
\end{aligned}$$

The Jacobian with respect to the feature state is given by:

$$\frac{\partial \mathbf{f}(\cdot)}{\partial \boldsymbol{\lambda}} = {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \begin{bmatrix} -\frac{1}{\rho}\sin(\theta)\sin(\phi) & \frac{1}{\rho}\cos(\theta)\cos(\phi) & -\frac{1}{\rho^2}\cos(\theta)\sin(\phi) \\ \frac{1}{\rho}\cos(\theta)\sin(\phi) & \frac{1}{\rho}\sin(\theta)\cos(\phi) & -\frac{1}{\rho^2}\sin(\theta)\sin(\phi) \\ 0 & -\frac{1}{\rho}\sin(\phi) & -\frac{1}{\rho^2}\cos(\phi) \end{bmatrix}$$

The Jacobians with respect to the anchor pose are:

$$\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^{I_a}_G\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top \left\lfloor {}^C_I\mathbf{R}^\top ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) \times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^G\mathbf{p}_{I_a}} &= \mathbf{I}_{3\times 3}
\end{aligned}$$

The Jacobians with respect to the IMU-camera extrinsics are:

$$\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C_I\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left\lfloor ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) \times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C\mathbf{p}_I} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top
\end{aligned}$$

5.6.5.6 Anchored Inverse Depth (MSCKF Version)

Note that a simpler version of inverse depth was used in the original MSCKF paper Mourikis and Roumeliotis [2007]. This representation does not have the singularity if it is represented in a camera frame the feature was measured from.

$$
\begin{aligned}
{}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda},\ {}^{I_a}_G\mathbf{R},\ {}^G\mathbf{p}_{I_a},\ {}^C_I\mathbf{R},\ {}^C\mathbf{p}_I) \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) + {}^G\mathbf{p}_{I_a} \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left(\frac{1}{\rho} \begin{bmatrix} \alpha \\ \beta \\ 1 \end{bmatrix} - {}^C\mathbf{p}_I \right) + {}^G\mathbf{p}_{I_a} \\
\text{where} \quad \boldsymbol{\lambda} &= \begin{bmatrix} \alpha & \beta & \rho \end{bmatrix}^\top
\end{aligned}
$$

The Jacobian with respect to the feature state is:

$$
\frac{\partial \mathbf{f}(\cdot)}{\partial \boldsymbol{\lambda}} = {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \begin{bmatrix} \frac{1}{\rho} & 0 & -\frac{1}{\rho^2}\alpha \\ 0 & \frac{1}{\rho} & -\frac{1}{\rho^2}\beta \\ 0 & 0 & -\frac{1}{\rho^2} \end{bmatrix}
$$

The Jacobians with respect to the anchor state are:

$$
\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^{I_a}_G\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top \left\lfloor {}^C_I\mathbf{R}^\top ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) \times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^G\mathbf{p}_{I_a}} &= \mathbf{I}_{3\times 3}
\end{aligned}
$$

The Jacobians with respect to the IMU-camera extrinsics are:

$$
\begin{aligned}
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C_I\mathbf{R}} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left\lfloor ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) \times \right\rfloor \\
\frac{\partial \mathbf{f}(\cdot)}{\partial {}^C\mathbf{p}_I} &= -{}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top
\end{aligned}
$$

5.6.5.7 Anchored Inverse Depth (MSCKF Single Depth Version)

This feature representation is based on the MSCKF representation Mourikis and Roumeliotis [2007], and the the single depth from VINS-Mono Qin et al. [2018]. As compared to the implementation in Qin et al. [2018], we are careful about how we handle treating of the bearing of the feature. During initialization we initialize a full 3D feature and then follow that by marginalize the bearing portion of it leaving the depth in the state vector. The marginalized bearing is then fixed for all future linearizations.

Then during update, we perform nullspace projection at every timestep to remove the feature dependence on this bearing. To do so, we need at least *two* sets of UV measurements to perform this bearing nullspace operation since we loose two dimensions of the feature in the process. We can define the feature measurement function as follows:

$$\begin{aligned}
{}^G\mathbf{p}_f &= \mathbf{f}(\boldsymbol{\lambda},\ {}^{I_a}_G\mathbf{R},\ {}^G\mathbf{p}_{I_a},\ {}^C_I\mathbf{R},\ {}^C\mathbf{p}_I) \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top ({}^{C_a}\mathbf{p}_f - {}^C\mathbf{p}_I) + {}^G\mathbf{p}_{I_a} \\
&= {}^{I_a}_G\mathbf{R}^\top {}^C_I\mathbf{R}^\top \left(\frac{1}{\rho}\hat{\mathbf{b}} - {}^C\mathbf{p}_I\right) + {}^G\mathbf{p}_{I_a} \\
\text{where} \quad \boldsymbol{\lambda} &= \begin{bmatrix}\rho\end{bmatrix}
\end{aligned}$$

In the above case we have defined a bearing $\hat{\mathbf{b}}$ which is the marginalized bearing of the feature after initialization. After collecting two measurement, we can nullspace project to remove the Jacobian in respect to this bearing variable.

5.7 Delayed Feature Initialization

We describe a method of delayed initialization of a 3D point feature as in `Visual-Inertial Odometry on Resource-Constrained Systems` Li [2014]. Specifically, given a set of measurements involving the state $\mathbf{x}$ and a new feature $\mathbf{f}$, we want to optimally and efficiently initialize the feature.

$$\mathbf{z}_i = \mathbf{h}_i\left(\mathbf{x}, \mathbf{f}\right) + \mathbf{n}_i$$

In general, we collect more than the minimum number of measurements at different times needed for initialization (i.e. delayed). For example, although in principle we need two monocular images to initialize a 3D point feature, we often collect more than two images in order to obtain better initialization. To process all collected measurements, we stack them and perform linearization around some linearization points (estimates) denoted by $\hat{\mathbf{x}}$ and $\hat{\mathbf{f}}$:

$$\begin{aligned}
\mathbf{z} = \begin{bmatrix}\mathbf{z}_1\\ \mathbf{z}_2\\ \vdots\\ \mathbf{z}_m\end{bmatrix} &= \mathbf{h}\left(\mathbf{x}, \mathbf{f}\right) + \mathbf{n} \\
\Rightarrow \quad \mathbf{r} = \mathbf{z} - \mathbf{h}(\hat{\mathbf{x}}, \hat{f}) &= \mathbf{H}_x\tilde{\mathbf{x}} + \mathbf{H}_f\tilde{\mathbf{f}} + \mathbf{n}
\end{aligned}$$

To efficiently compute the resulting augmented covariance matrix, we perform `Givens rotations` to zero-out rows in $\mathbf{H}_f$ with indices larger than the dimension of $\tilde{\mathbf{f}}$, and apply the same Givens rotations to $\mathbf{H}_x$ and $\mathbf{r}$. As a result of this operation, we have the following linear system:

$$\begin{bmatrix}\mathbf{r}_1\\ \mathbf{r}_2\end{bmatrix} = \begin{bmatrix}\mathbf{H}_{x1}\\ \mathbf{H}_{x2}\end{bmatrix}\tilde{\mathbf{x}} + \begin{bmatrix}\mathbf{H}_{f1}\\ \mathbf{0}\end{bmatrix}\tilde{\mathbf{f}} + \begin{bmatrix}\mathbf{n}_1\\ \mathbf{n}_2\end{bmatrix}$$

Note that the bottom system essentially is corresponding to the nullspace projection as in the MSCKF update and $\mathbf{H}_{f1}$ is generally invertible. Note also that we assume the measurement noise is isotropic; otherwise, we should first perform whitening to make it isotropic, which would save significant computations. So, if the original measurement noise covariance $\mathbf{R} = \sigma^2\mathbf{I}_m$ and the dimension of $\tilde{\mathbf{f}}$ is n, then the inferred measurement noise covariance will be $\mathbf{R}_1 = \sigma^2\mathbf{I}_n$ and $\mathbf{R}_2 = \sigma^2\mathbf{I}_{m-n}$.

Now we can directly solve for the error of the new feature based on the first subsystem:

$$\begin{aligned} \tilde{\mathbf{f}} &= \mathbf{H}_{f1}^{-1}(\mathbf{r}_1 - \mathbf{n}_1 - \mathbf{H}_x\tilde{\mathbf{x}}) \\ \Rightarrow\ \mathbb{E}[\tilde{\mathbf{f}}] &= \mathbf{H}_{f1}^{-1}(\mathbf{r}_1) \end{aligned}$$

where we assumed noise and state error are zero mean. We can update $\hat{\mathbf{f}}$ with this correction by $\hat{\mathbf{f}} + \mathbb{E}[\tilde{\mathbf{f}}]$. Note that this is equivalent to a Gauss Newton step for solving the corresponding maximum likelihood estimation (MLE) formed by fixing the estimate of $\mathbf{x}$ and optimizing over the value of $\hat{\mathbf{f}}$, and should therefore be zero if we used such an optimization to come up with our initial estimate for the new variable.

We now can compute the covariance of the new feature as follows:

$$\begin{aligned} \mathbf{P}_{ff} &= \mathbb{E}\Big[(\tilde{\mathbf{f}} - \mathbb{E}[\tilde{\mathbf{f}}])(\tilde{\mathbf{f}} - \mathbb{E}[\tilde{\mathbf{f}}])^\top\Big] \\ &= \mathbb{E}\Big[(\mathbf{H}_{f1}^{-1}(-\mathbf{n}_1 - \mathbf{H}_{x1}\tilde{\mathbf{x}}))(\mathbf{H}_{f1}^{-1}(-\mathbf{n}_1 - \mathbf{H}_{x1}\tilde{\mathbf{x}}))^\top\Big] \\ &= \mathbf{H}_{f1}^{-1}(\mathbf{H}_{x1}\mathbf{P}_{xx}\mathbf{H}_{x1}^\top + \mathbf{R}_1)\mathbf{H}_{f1}^{-\top} \end{aligned}$$

and the cross correlation can be computed as:

$$\begin{aligned} \mathbf{P}_{xf} &= \mathbb{E}\Big[(\tilde{\mathbf{x}})(\tilde{\mathbf{f}} - \mathbb{E}[\tilde{\mathbf{f}}])^\top\Big] \\ &= \mathbb{E}\Big[(\tilde{\mathbf{x}})(\mathbf{H}_{f1}^{-1}(-\mathbf{n}_1 - \mathbf{H}_{x1}\tilde{\mathbf{x}}))^\top\Big] \\ &= -\mathbf{P}_{xx}\mathbf{H}_{x1}^\top\mathbf{H}_{f1}^{-\top} \end{aligned}$$

These entries can then be placed in the correct location for the covariance. For example when initializing a new feature to the end of the state, the augmented covariance would be:

$$\mathbf{P}_{aug} = \begin{bmatrix}\mathbf{P}_{xx} & \mathbf{P}_{xf}\\ \mathbf{P}_{xf}^\top & \mathbf{P}_{ff}\end{bmatrix}$$

Note that this process does not update the estimate for $\mathbf{x}$. However, after initialization, we can then use the second system, $\mathbf{r}_2$, $\mathbf{H}_{x2}$, and $\mathbf{n}_2$ to update our new state through a standard EKF update (see Linear Measurement Update section).

5.8 MSCKF Nullspace Projection

In the standard EKF update, given a linearized measurement error (or residual) equation:

$$\tilde{\mathbf{z}}_{m,k} \simeq \mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f + \mathbf{n}_k$$

we naively need to compute the residual covariance matrix $\mathbf{P}_{zz}$ as follows:

$$\begin{aligned}
\mathbf{P}_{zz} &= \mathbb{E}\left[\tilde{\mathbf{z}}_{m,k} m, k \tilde{\mathbf{z}}_{m,k}^\top\right] \\
&= \mathbb{E}\left[(\mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f + \mathbf{n}_k)(\mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f + \mathbf{n}_k)^\top\right] \\
&= \mathbb{E}\Big[\mathbf{H}_x \tilde{\mathbf{x}}_k \tilde{\mathbf{x}}_k^\top \mathbf{H}_x^\top + \mathbf{H}_x \tilde{\mathbf{x}}_k {}^G\tilde{\mathbf{p}}_f^\top \mathbf{H}_f^\top + \mathbf{H}_x \tilde{\mathbf{x}}_k \mathbf{n}_k^\top \\
&\qquad + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f \tilde{\mathbf{x}}_k^\top \mathbf{H}_x^\top + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f {}^G\tilde{\mathbf{p}}_f^\top \mathbf{H}_f^\top + \mathbf{H}_f {}^G\tilde{\mathbf{p}}_f \mathbf{n}_k^\top \\
&\qquad + \mathbf{n}_k \tilde{\mathbf{x}}_k^\top \mathbf{H}_x^\top + \mathbf{n}_k {}^G\tilde{\mathbf{p}}_f^\top \mathbf{H}_f^\top + \mathbf{n}_k \mathbf{n}_k^\top\Big] \\
&= \mathbf{H}_x \mathbb{E}\Big[\tilde{\mathbf{x}}_k \tilde{\mathbf{x}}_k^\top\Big] \mathbf{H}_x^\top + \mathbf{H}_x \mathbb{E}\Big[\tilde{\mathbf{x}}_k {}^G\tilde{\mathbf{p}}_f^\top\Big] \mathbf{H}_f^\top + \mathbf{H}_f \mathbb{E}\Big[{}^G\tilde{\mathbf{p}}_f \tilde{\mathbf{x}}_k^\top\Big] \mathbf{H}_x^\top + \mathbf{H}_f \mathbb{E}\Big[{}^G\tilde{\mathbf{p}}_f {}^G\tilde{\mathbf{p}}_f^\top\Big] \mathbf{H}_f^\top \\
&\qquad + \mathbf{H}_f \mathbb{E}\Big[{}^G\tilde{\mathbf{p}}_f \mathbf{n}_k^\top\Big] + \mathbb{E}\Big[\mathbf{n}_k {}^G\tilde{\mathbf{p}}_f^\top\Big] \mathbf{H}_f^\top + \mathbb{E}\Big[\mathbf{n}_k \mathbf{n}_k^\top\Big] \\
&= \mathbf{H}_x \mathbf{P}_{xx} \mathbf{H}_x^\top + \mathbf{H}_x \mathbf{P}_{xf} \mathbf{H}_f^\top + \mathbf{H}_f \mathbf{P}_{fx} \mathbf{H}_x^\top + \mathbf{H}_f \mathbf{P}_{ff} \mathbf{H}_f^\top \\
&\qquad + \mathbf{H}_f \mathbf{P}_{fn} + \mathbf{P}_{nf} \mathbf{H}_f^\top + \mathbf{R}_d
\end{aligned}$$

However, there would be a big problem in visual-inertial odometry (VIO); that is, we do not know what the prior feature covariance and it is coupled with both the state, itself, and the noise (i.e., $\mathbf{P}_{xf}$, $\mathbf{P}_{ff}$, and $\mathbf{P}_{nf}$). This motivates the need for a method to remove the feature ${}^G\tilde{\mathbf{p}}_f$ from the linearized measurement equation (thus removing the correlation between the measurement and its error).

To this end, we start with the measurement residual function by removing the "sensitivity" to feature error we compute and apply the left nullspace of the Jacobian $\mathbf{H}_f$. We can compute it using QR decomposition as follows:

$$\mathbf{H}_f = \begin{bmatrix} \mathbf{Q_1} & \mathbf{Q_2} \end{bmatrix} \begin{bmatrix} \mathbf{R_1} \\ \mathbf{0} \end{bmatrix} = \mathbf{Q_1} \mathbf{R_1}$$

Multiplying the linearized measurement equation by the nullspace of the feature Jacobian from the left yields:

$$\begin{aligned}
\tilde{\mathbf{z}}_{m,k} &\simeq \mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{Q_1} \mathbf{R_1} {}^G\tilde{\mathbf{p}}_f + \mathbf{n}_k \\
\Rightarrow\ \mathbf{Q_2}^\top \tilde{\mathbf{z}}_m &\simeq \mathbf{Q_2}^\top \mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{Q_2}^\top \mathbf{Q_1} \mathbf{R_1} {}^G\tilde{\mathbf{p}}_f + \mathbf{Q_2}^\top \mathbf{n}_k \\
\Rightarrow\ \mathbf{Q_2}^\top \tilde{\mathbf{z}}_m &\simeq \mathbf{Q_2}^\top \mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{Q_2}^\top \mathbf{n}_k \\
\Rightarrow\ \tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{H}_{o,k} \tilde{\mathbf{x}}_k + \mathbf{n}_{o,k}
\end{aligned}$$

where we have employed the fact that $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are orthonormal.

We now examine the dimensions of the involved matrices to appreciate the computation saving gained from this nullspace projection.

$$\begin{aligned}
\text{size}(\mathbf{H}_f) &= 2n \times 3 \ \text{ where } n \text{ is the number of uv measurements of this feature} \\
\text{size}({}^G\tilde{\mathbf{p}}_f) &= 3 \times 1 \\
\text{size}(\mathbf{H}_x) &= 2n \times 15 + 6c \ \text{ where } c \text{ is the number of clones} \\
\text{size}(\tilde{\mathbf{x}}_k) &= 15 + 6c \times 1 \ \text{ where } c \text{ is the number of clones} \\
\text{rank}(\mathbf{H}_f) &\leq \min(2n, 3) = 3 \ \text{ where equality holds in most cases} \\
\text{nullity}(\mathbf{H}_f) &= \text{size}(\mathbf{x}) - \text{rank}(\mathbf{H}_f) = 2n - 3 \ \text{ assuming full rank}
\end{aligned}$$

With that, we can have the following conclusion about the sizes when the nullspace is applied:

$$\begin{aligned}
\mathbf{Q_2}^\top \tilde{\mathbf{z}}_{m,k} &\simeq \mathbf{Q_2}^\top \mathbf{H}_x \tilde{\mathbf{x}}_k + \mathbf{Q_2}^\top \mathbf{n}_k \\
\Rightarrow \ (2n - 3 \times 2n)(2n \times 1) &= (2n - 3 \times 2n)(2n \times 15 + 6c)(15 + 6c \times 1) \\
&\quad + (2n - 3 \times 2n)(2n \times 1) \\
\tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{H}_{o,k} \tilde{\mathbf{x}}_k + \mathbf{n}_o \\
\Rightarrow \ (2n - 3 \times 1) &= (2n - 3 \times 15 + 6c)(15 + 6c \times 1) + (2n - 3 \times 1)
\end{aligned}$$

Finally, we perform the EKF update using the inferred measurement $\mathbf{z}_{o,k}$:

$$\begin{aligned}
\hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + \mathbf{P}_{k|k-1} \mathbf{H}_{o,k}^\top (\mathbf{H}_{o,k} \mathbf{P}_{k|k-1} \mathbf{H}_{o,k}^\top + \mathbf{R}_o)^{-1} \tilde{\mathbf{z}}_{o,k} \\
\mathbf{P}_{k|k} &= \mathbf{P}_{k|k-1} - \mathbf{P}_{k|k-1} \mathbf{H}_{o,k}^\top (\mathbf{H}_{o,k} \mathbf{P}_{k|k-1} \mathbf{H}_{o,k}^\top + \mathbf{R}_o)^{-1} \mathbf{H}_{o,k} \mathbf{P}_{k|k-1}^\top
\end{aligned}$$

where the time index (subscript) $k|k-1$ refers to the prior estimate which was denoted before by symbol $\ominus$ and $k|k$ corresponds to the posterior (or updated) estimate indicated before by $\oplus$.

5.8.1 Implementation

Using Eigen 3 library, we perform QR decomposition to get the nullspace. Here we know that the size of $\mathbf{Q}_1$ is $2n \times 3$, which corresponds to the number of observations and size of the 3D point feature state.

```
Eigen::ColPivHouseholderQR<Eigen::MatrixXd> qr(H_f.rows(), H_f.cols());
qr.compute(H_f);
Eigen::MatrixXd Q = qr.householderQ();
Eigen::MatrixXd Q1 = Q.block(0,0,Q.rows(),3);
Eigen::MatrixXd Q2 = Q.block(0,3,Q.rows(),Q.cols()-3);
```

5.9 Measurement Compression

One of the most costly opeerations in the EKF update is the matrix multiplication. To mitigate this issue, we perform the thin QR decomposition of the measurement Jacobian after nullspace projection:

$$\mathbf{H}_{o,k} = \begin{bmatrix} \mathbf{Q_1} & \mathbf{Q_2} \end{bmatrix} \begin{bmatrix} \mathbf{R_1} \\ \mathbf{0} \end{bmatrix} = \mathbf{Q}_1\mathbf{R}_1$$

This QR decomposition can be performed again using `Givens rotations` (note that this operation in general is not cheap though). We apply this QR to the linearized measurement residuals to compress measurements:

$$\begin{aligned}
\tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{H}_{o,k}\tilde{\mathbf{x}}_k + \mathbf{n}_o \\
\tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{Q}_1\mathbf{R}_1\tilde{\mathbf{x}}_k + \mathbf{n}_o \\
\mathbf{Q_1}^\top\tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{Q_1}^\top\mathbf{Q_1}\mathbf{R_1}\tilde{\mathbf{x}}_k + \mathbf{Q_1}^\top\mathbf{n}_o \\
\mathbf{Q_1}^\top\tilde{\mathbf{z}}_{o,k} &\simeq \mathbf{R_1}\tilde{\mathbf{x}}_k + \mathbf{Q_1}^\top\mathbf{n}_o \\
\Rightarrow\ \tilde{\mathbf{z}}_{n,k} &\simeq \mathbf{H}_{n,k}\tilde{\mathbf{x}}_k + \mathbf{n}_n
\end{aligned}$$

As a result, the compressed measurement Jacobian will be of the size of the state, which will signficantly reduce the EKF update cost:

$$\begin{aligned}
\hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + \mathbf{P}_{k|k-1}\mathbf{H}_{n,k}^\top(\mathbf{H}_{n,k}\mathbf{P}_{k|k-1}\mathbf{H}_{n,k}^\top + \mathbf{R}_n)^{-1}\tilde{\mathbf{z}}_{n,k} \\
\mathbf{P}_{k|k} &= \mathbf{P}_{k|k-1} - \mathbf{P}_{k|k-1}\mathbf{H}_{n,k}^\top(\mathbf{H}_{n,k}\mathbf{P}_{k|k-1}\mathbf{H}_{n,k}^\top + \mathbf{R}_n)^{-1}\mathbf{H}_{n,k}\mathbf{P}_{k|k-1}^\top
\end{aligned}$$

5.10 Zero Velocity Update

The key idea of the zero velocity update (ZUPT) is to allow for the system to reduce its uncertainty leveraging motion knowledge (i.e. leverage the fact that the system is stationary). This is of particular importance in cases where we have a monocular system without any temporal SLAM features. In this case, if we are stationary we will be unable to triangulate features and thus will be unable to update the system. This can be avoided by either using a stereo system or temporal SLAM features. One problem that both of these don't solve is the issue of dynamic environmental objects. In a typical autonomous car scenario the sensor system will become stationary at stop lights in which dynamic objects, such as other cars crossing the intersection, can quickly corrupt the system. A zero velocity update and skipping feature tracking can address these issues if we are able to classify the cases where the sensor system is at rest.

5.10.1 Constant Velocity Synthetic Measurement

To perform update, we create a synthetic "measurement" which says that the current **true** acceleration and angular velocity is zero. As compared to saying the velocity is zero, we can model the uncertainty of these measurements based on the readings from our inertial measurement unit.

$$\begin{aligned}\mathbf{a} &= \mathbf{0} \\ \boldsymbol{\omega} &= \mathbf{0}\end{aligned}$$

It is important to realize this is not strictly enforcing zero velocity, but really a constant velocity. This means we can have a false detection at constant velocity times (zero acceleration), but this can be easily addressed by a velocity magnitude check. We have the following measurement equation relating this above synthetic "measurement" to the currently recorded inertial readings:

$$\begin{aligned}\mathbf{a} &= \mathbf{a}_m - \mathbf{b}_a - {}^{I_k}_{G}\mathbf{R}^G\mathbf{g} - \mathbf{n}_a \\ \boldsymbol{\omega} &= \boldsymbol{\omega}_m - \mathbf{b}_g - \mathbf{n}_g\end{aligned}$$

It is important to note that here our actual measurement is the true $\mathbf{a}$ and $\boldsymbol{\omega}$ and thus we will have the following residual where we will subtract the synthetic "measurement" and our measurement function:

$$\tilde{\mathbf{z}} = \begin{bmatrix} \mathbf{a} - \left(\mathbf{a}_m - \mathbf{b}_a - {}^{I_k}_{G}\mathbf{R}^G\mathbf{g} - \mathbf{n}_a\right) \\ \boldsymbol{\omega} - \left(\boldsymbol{\omega}_m - \mathbf{b}_g - \mathbf{n}_g\right) \end{bmatrix} \qquad = \begin{bmatrix} -\left(\mathbf{a}_m - \mathbf{b}_a - {}^{I_k}_{G}\mathbf{R}^G\mathbf{g} - \mathbf{n}_a\right) \\ -\left(\boldsymbol{\omega}_m - \mathbf{b}_g - \mathbf{n}_g\right) \end{bmatrix}$$

Where we have the following Jacobians in respect to our state:

$$\begin{aligned}\frac{\partial \tilde{\mathbf{z}}}{\partial {}^{I_k}_{G}\mathbf{R}} &= -\left\lfloor {}^{I_k}_{G}\mathbf{R}^G\mathbf{g}\times \right\rfloor \\ \frac{\partial \tilde{\mathbf{z}}}{\partial \mathbf{b}_a} &= \frac{\partial \tilde{\mathbf{z}}}{\partial \mathbf{b}_g} = -\mathbf{I}_{3\times 3}\end{aligned}$$

5.10.2 Zero Velocity Detection

Zero velocity detection in itself is a challenging problem which has seen many different works tried to address this issue Wagstaff et al. [2017], Ramanandan et al. [2011], Davidson et al. [2009]. Most works boil down to simple thresholding and the approach is to try to determine the optimal threshold which allows for the best classifications of zero velocity update (ZUPT) portion of the trajectories. There have been other works, Wagstaff et al. [2017] and Ramanandan et al. [2011], which have looked at more complicated methods and try to address the issue that this threshold can be dependent on the type of different motions (such as running vs walking) and characteristics of the platform which the sensor is mounted on (we want to ignore vehicle engine vibrations and other non-essential observed vibrations).

5.10.2.1 Inertial-based Detection

We approach this detection problem based on tuning of a χ^2, chi-squared, thresholding based on the measurement model above. It is important to note that we also have a velocity magnitude check which is aimed at preventing constant velocity cases which have non-zero magnitude. More specifically, we perform the following threshold check to see if we are current at zero velocity:

$$\tilde{\mathbf{z}}^\top(\mathbf{H}\mathbf{P}\mathbf{H}^\top + \alpha\mathbf{R})^{-1}\tilde{\mathbf{z}} < \chi^2$$

We found that in the real world experiments, typically the inertial measurement noise $\mathbf{R}$ needs to be inflated by $\alpha \in [50, 100]$ times to allow for proper detection. This can hint that we are using overconfident inertial noises, or that there are additional frequencies (such as the vibration of motors) which inject additional noises.

5.10.2.2 Disparity-based Detection

We additionally have a detection method which leverages the visual feature tracks. Given two sequential images, the assumption is if there is very little disparity change between feature tracks then we will be stationary. Thus we calculate the average disparity and threshold on this value.

$$\frac{1}{N}\sum_{i=0}^{N}||\mathbf{uv}_{k,i} - \mathbf{uv}_{k-1,i}|| < \Delta d$$

This seems to work reasonably well, but can fail if the environment is dynamic in nature, thus it can be useful to use both the inertial and disparity-based methods together in very dynamic environments.

Chapter 6

Visual-Inertial Simulator

6.1 B-Spline Interpolation

At the center of the simulator is an $\mathbb{SE}(3)$ b-spline which allows for the calculation of the pose, velocity, and accelerations at any given timestep along a given trajectory. We follow the work of Mueggler et al. Mueggler et al. [2018] and Patron et al. Patron-Perez et al. [2015] in which given a series of uniformly distributed "control points" poses the pose $\{S\}$ at a given timestep t_s can be interpolated by:

$$
\begin{aligned}
{}^{G}_{S}\mathbf{T}(u(t_s)) &= {}^{G}_{i-1}\mathbf{T}\ \mathbf{A}_0\ \mathbf{A}_1\ \mathbf{A}_2 \\
\mathbf{A}_j &= \exp\Big(B_j(u(t))\ {}^{i-1+j}_{i+j}\mathbf{\Omega}\Big) \\
{}^{i-1}_{i}\mathbf{\Omega} &= \log\Big({}^{G}_{i-1}\mathbf{T}^{-1}\ {}^{G}_{i}\mathbf{T}\Big) \\
B_0(u(t)) &= \frac{1}{3!}\ (5+3u-3u^2+u^3) \\
B_1(u(t)) &= \frac{1}{3!}\ (1+3u+3u^2-2u^3) \\
B_2(u(t)) &= \frac{1}{3!}\ (u^3)
\end{aligned}
$$

where $u(t_s) = (t_s - t_i)/(t_{i+1} - t_i)$, $\exp(\cdot)$, $\log(\cdot)$ are the $\mathbb{SE}(3)$ matrix exponential ov_core::exp_se3 and logarithm ov_core::log_se3. The frame notations can be seen in the above figure and we refer the reader to the ov_core::BsplineSE3 class for more details. The above equation can be interpretative as compounding the fractions portions of the bounding poses to the first pose ${}^{G}_{i-1}\mathbf{T}$. From this above equation, it is simple to take the derivative in respect to time, thus allowing the computation of the velocity and acceleration at any point.

The only needed input into the simulator is a pose trajectory which we will then uniformly sample to construct control points for this spline. This spline is then used to both generate the inertial measurements while also providing the pose information needed to generate visual-bearing measurements.

6.2 Inertial Measurements

To incorporate inertial measurements from a IMU sensor, we can leverage the continuous nature and C^2-continuity of our cubic B-spline. We can define the sensor measurement from a IMU as follows:

$$\begin{aligned}
{}^I\boldsymbol{\omega}_m(t) &= {}^I\boldsymbol{\omega}(t) + \mathbf{b}_\omega + \mathbf{n}_\omega \\
{}^I\mathbf{a}_m(t) &= {}^I\mathbf{a}(t) + {}^{I(t)}_{G}\mathbf{R}\,{}^G\mathbf{g} + \mathbf{b}_a + \mathbf{n}_a \\
\dot{\mathbf{b}}_\omega &= \mathbf{n}_{wg} \\
\dot{\mathbf{b}}_a &= \mathbf{n}_{wa}
\end{aligned}$$

where each measurement is corrupted with some white noise and random-walk bias. To obtain the true measurements from our $\mathbb{SE}(3)$ b-spline we can do the following:

$$\begin{aligned}
{}^I\boldsymbol{\omega}(t) &= \mathrm{vee}\Big({}^G_I\mathbf{R}(u(t))^\top\,{}^G_I\dot{\mathbf{R}}(u(t))\Big) \\
{}^I\mathbf{a}(t) &= {}^G_I\mathbf{R}(u(t))^\top\,{}^G\ddot{\mathbf{p}}_I(u(t))
\end{aligned}$$

where $\mathrm{vee}(\cdot)$ returns the vector portion of the skew-symmetric matrix (see ov_core::vee). These are then corrupted using the random walk biases and corresponding white noises. For example we have the following:

$$\begin{aligned}
\omega_m(t) &= \omega(t) + b_\omega(t) + \sigma_w \frac{1}{\sqrt{\Delta t}} \mathrm{gennoise}(0,1) \\
b_\omega(t+\Delta t) &= b_\omega(t) + \sigma_{wg}\sqrt{\Delta t}\ \mathrm{gennoise}(0,1) \\
t &= t + \Delta t
\end{aligned}$$

Note that this is repeated per-scalar value as compared to the vector and identically for the accelerometer readings. The $\mathrm{gennoise}(m,v)$ function generates a random scalar float with mean *m* and variance *v*. The Δt is our sensor sampling rate that we advance time forward with.

6.3 Visual-Bearing Measurement

The first step that we perform after creating the b-spline trajectory is the generation of a map of point features. To generate these features, we increment along the spline at a fixed interval and ensure that all cameras see enough features in the map. If there are not enough features in the given frame, we generate new features by sending random rays from the camera out and assigning a random depth. This feature is then added to the map so that it can be projected into future frames.

After the map generation phase, we generate feature measurements by projecting them into the current frame. Projected features are limited to being with-in the field of view of the camera, in front of the camera, and close in distance. Pixel noise can be directly added to the raw pixel values.

Chapter 7

System Evaluation

The goal of our evaluation is to ensure fair comparison to other methods and our own. The actual metrics we use can be found on the Filter Evaluation Metrics page. Using our metrics we wish to provide insight into *why* our method does better and in what ways (as no method will outperform in all aspects). Since we are also interested in applying the systems to real robotic applications, the realtime performance is also a key metric we need to investigate. Timing of different system components is also key to removing bottlenecks and seeing where performance improvements or estimator approximations might help reduce complexity.

The key metrics we are interested in evaluating are the following:

- Absolute Trajectory Error (ATE)
- Relative Pose Error (RPE)
- Root Mean Squared Error (RMSE)
- Normalized Estimation Error Squared (NEES)
- Estimator Component Timing
- System Hardware Usage (memory and computation)

7.1 System Evaluation Guides

- Filter Evaluation Metrics — Definitions of different metrics for estimator accuracy.
- Filter Error Evaluation Methods — Error evaluation methods for evaluating system performance.
- Filter Timing Analysis — Timing of estimator components and complexity.

7.2 Filter Evaluation Metrics

7.2.1 Absolute Trajectory Error (ATE)

The Absolute Trajectory Error (ATE) is given by the simple difference between the estimated trajectory and groundtruth after it has been aligned so that it has minimal error. First the "best" transform between the groundtruth and estimate is computed, afterwhich the error is computed at every timestep and then averaged. We recommend reading Zhang and Scaramuzza Zhang and Scaramuzza [2018] paper for details. For a given dataset with N runs of the same algorithm with K pose measurements, we can compute the following for an aligned estimated trajectory $\hat{\mathbf{x}}^+$:

$$e_{ATE} = \frac{1}{N}\sum_{i=1}^{N}\sqrt{\frac{1}{K}\sum_{k=1}^{K}||\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}^+_{k,i}||^2_2}$$

7.2.2 Relative Pose Error (RPE)

The Relative Pose Error (RPE) is calculated for segments of the dataset and allows for introspection of how localization solutions drift as the length of the trajectory increases. The other key advantage over ATE error is that it is less sensitive to jumps in estimation error due to sampling the trajectory over many smaller segments. This allows for a much fairer comparision of methods and is what we recommend all authors publish results for. We recommend reading Zhang and Scaramuzza Zhang and Scaramuzza [2018] paper for details. We first define a set of segment lengths $\mathcal{D} = [d_1,\ d_2, \cdots,\ d_V]$ which we compute the relative error for. We can define the relative error for a trajectory split into D_i segments of d_i length as follows:

$$\tilde{\mathbf{x}}_r = \mathbf{x}_k \boxminus \mathbf{x}_{k+d_i}$$
$$e_{rpe,d_i} = \frac{1}{D_i}\sum_{k=1}^{D_i}||\tilde{\mathbf{x}}_r \boxminus \hat{\tilde{\mathbf{x}}}_r||^2_2$$

7.2.3 Root Mean Squared Error (RMSE)

When evaluating a system on a *single* dataset is the Root Mean Squared Error (RMSE) plots. This plots the RMSE at every timestep of the trajectory and thus can provide insight into timesteps where the estimation performance suffers. For a given dataset with N runs of the same algorithm we can compute the following at each timestep k:

$$e_{rmse,k} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}||\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i}||^2_2}$$

7.2.4 Normalized Estimation Error Squared (NEES)

Normalized Estimation Error Squared (NEES) is a standard way to characterize if the estimator is being consistent or not. In general NEES is just the normalized error which should be the degrees of freedoms of the state variables. Thus in the case of position and orientation we should get a NEES of three at every timestep. To compute the average NEES for a dataset with N runs of the same algorithm we can compute the following at each timestep k:

$$e_{nees,k} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i})^\top \mathbf{P}_{k,i}^{-1}(\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i})$$

7.2.5 Single Run Consistency

When looking at a *single run* and wish to see if the system is consistent it is interesting to look a its error in respect to its estimated uncertainty. Specifically we plot the error and the estimator 3σ bound. This provides insight into if the estimator is becoming over confident at certain timesteps. Note this is for each component of the state, thus we need to plot x,y,z and orientation independently. We can directly compute the error at timestep k:

$$\begin{aligned}\mathbf{e}_k &= \mathbf{x}_k \boxminus \hat{\mathbf{x}}_k \\ &\text{where } \ \mathbf{e}_k \sim \mathcal{N}(0, \mathbf{P})\end{aligned}$$

7.3 Filter Error Evaluation Methods

Installation Warning

If you plan to use the included plotting from the cpp code, you will need to make sure that you have matplotlib and python 2.7 installed. We use the to `matplotlib-cpp` to call this external library and generate the desired figures. Please see Additional Evaluation Requirements for more details on the exact install.

7.3.1 Collection

The first step in any evaluation is to first collect the estimated trajectory of the proposed systems. Since we are interested in robotic application of our estimators we want to record the estimate at the current timestep (as compared to a "smoothed" output or one that includes loop-closures from future timesteps). Within the ROS framework, this means that we just need to publish the current estimate at the current timestep. We recommend using the following ov_eval::Recorder utility for recording the estimator output directly into a text file. Works with topics of type `Pose↩ WithCovarianceStamped`, `PoseStamped`, `TransformStamped`, and `Odometry`.

```
<node name="recorder_estimate" pkg="ov_eval" type="pose_to_file" output="screen">
    <param name="topic"      type="str" value="/ov_msckf/poseimu" />
    <param name="topic_type" type="str" value="PoseWithCovarianceStamped" />
    <param name="output"     type="str" value="/home/user/data/traj_log.txt" />
</node>
```

7.3.2 Transformation

We now need to ensure both our estimated trajectory and groundtruth are in the correct formats for us to read in. We need things to be in the RPE text file format (i.e. time(s), px, py, pz, qx, qy, qz, qw). We have a nice helper script that will transform ASL / EuRoC groundtruth files to the correct format. By default the EuRoC groundtruth has the timestamp in nanoseconds and the quaternion is in an incorrect order (i.e. time(ns), px, py, pz, qw, qx, qy, qz). A user can either process all CSV files in a given folder, or just a specific one.

```
rosrun ov_eval format_convert folder/path/
rosrun ov_eval format_convert file.csv
```

In addition we have a specific folder structure that is assumed. We store trajectories by first their algorithm name and then a folder for each dataset this algorithm was run on. The folder names of the datasets need to match the groundtruth trajectory files which should be in their own separate folder. Please see the example recorded datasets for how to structure your folders.

```
truth/
    dateset_name_1.txt
    dateset_name_2.txt
algorithms/
    open_vins/
        dataset_name_1/
            run1.txt
            run2.txt
            run3.txt
        dataset_name_2/
            run1.txt
            run2.txt
            run3.txt
    okvis_stereo/
        dataset_name_1/
            run1.txt
            run2.txt
            run3.txt
        dataset_name_2/
            run1.txt
            run2.txt
            run3.txt
    vins_mono/
        dataset_name_1/
            run1.txt
            run2.txt
            run3.txt
        dataset_name_2/
            run1.txt
            run2.txt
            run3.txt
```

7.3.3 Processing & Plotting

Now that we have our data recorded and in the correct format we can now work on processing and plotting it. In the next few sections we detail how to do this for absolute trajectory error, relative pose error, normalized estimation error squared, and bounded root mean squared error plots. We will first process the data into a set of output text files which a user can then use to plot the results in their program or language of choice. The align mode of all the following commands can be of type `posyaw`, `posyawsingle`, `se3`, `se3single`, `sim3`, and `none`.

7.3.3.1 Script "plot_trajectories"

To plot the data we can us the following command which will plot a 2d xy and z-time position plots. It will use the filename as the name in the legend, so you can change that to change the legend or edit the code.

```
rosrun ov_eval plot_trajectories <align_mode> <file_gt.txt> ... <file_est9.txt>
rosrun ov_eval plot_trajectories posyaw 1565371553_estimate.txt truths/V1_01_easy.txt
```

7.3.3.2 Script "error_singlerun"

The single run script will plot statistics and also 3 σ bounds if available. One can use this to see consistency of the estimator or debug how the current run has gone. It also reports to console the average RMSE and RPE values for this run along with the number of samples. To change the RPE distances you will need to edit the code currently.

```
rosrun ov_eval error_singlerun <align_mode> <file_gt.txt> <file_est.txt>
rosrun ov_eval error_singlerun posyaw 1565371553_estimate.txt truths/V1_01_easy.txt
```

Example output:

```
[ INFO] [1583422165.069376426]: [COMP]: 2813 poses in 1565371553_estimate => length of 57.36 meters
[ INFO] [1583422165.091423722]: ======================================
[ INFO] [1583422165.091438299]: Absolute Trajectory Error
[ INFO] [1583422165.091445338]: ======================================
[ INFO] [1583422165.091453099]: rmse_ori = 0.677 | rmse_pos = 0.055
[ INFO] [1583422165.091459679]: mean_ori = 0.606 | mean_pos = 0.051
[ INFO] [1583422165.091466321]: min_ori  = 0.044 | min_pos  = 0.001
[ INFO] [1583422165.091474211]: max_ori  = 1.856 | max_pos  = 0.121
[ INFO] [1583422165.091481730]: std_ori  = 0.302 | std_pos  = 0.021
[ INFO] [1583422165.127869924]: ======================================
[ INFO] [1583422165.127891080]: Relative Pose Error
[ INFO] [1583422165.127898322]: ======================================
[ INFO] [1583422165.127908551]: seg 8 - median_ori = 0.566 | median_pos = 0.068 (2484 samples)
[ INFO] [1583422165.127919603]: seg 16 - median_ori = 0.791 | median_pos = 0.060 (2280 samples)
[ INFO] [1583422165.127927646]: seg 24 - median_ori = 0.736 | median_pos = 0.070 (2108 samples)
[ INFO] [1583422165.127934904]: seg 32 - median_ori = 0.715 | median_pos = 0.071 (1943 samples)
[ INFO] [1583422165.127942178]: seg 40 - median_ori = 0.540 | median_pos = 0.063 (1792 samples)
```

7.3.3.3 Script "error_dataset"

This dataset script will evaluate how a series of algorithms compare on a single dataset. Normally this is used if you just have single dataset you want to compare algorithms on, or compare a bunch variations of your algorithm to a simulated trajectory. In the console it will output the ATE 3D and 2D, along with the 3D RPE and 3D NEES for each method after it performs alignment. To change the RPE distances you will need to edit the code currently.

```
rosrun ov_eval error_dataset <align_mode> <file_gt.txt> <folder_algorithms>
rosrun ov_eval error_dataset posyaw truths/V1_01_easy.txt algorithms/
```

Example output:

```
[ INFO] [1583422654.333642977]: ======================================
[ INFO] [1583422654.333915102]: [COMP]: processing mono_ov_slam algorithm
[ INFO] [1583422654.362655719]: [TRAJ]: q_ESTtoGT = 0.000, 0.000, -0.129, 0.992 | p_ESTinGT = 0.978, 2.185,
       0.932 | s = 1.00
....
[ INFO] [1583422654.996859432]: [TRAJ]: q_ESTtoGT = 0.000, 0.000, -0.137, 0.991 | p_ESTinGT = 0.928, 2.166,
       0.957 | s = 1.00
[ INFO] [1583422655.041009388]:     ATE: mean_ori = 0.684 | mean_pos = 0.057
[ INFO] [1583422655.041031730]:     ATE: std_ori  = 0.14938 | std_pos  = 0.01309
[ INFO] [1583422655.041039552]:     ATE 2D: mean_ori = 0.552 | mean_pos = 0.053
[ INFO] [1583422655.041046362]:     ATE 2D: std_ori  = 0.17786 | std_pos  = 0.01421
[ INFO] [1583422655.044187033]:     RPE: seg 7 - mean_ori = 0.543 | mean_pos = 0.065 (25160 samples)
[ INFO] [1583422655.047047771]:     RPE: seg 14 - mean_ori = 0.593 | mean_pos = 0.060 (23470 samples)
[ INFO] [1583422655.049672955]:     RPE: seg 21 - mean_ori = 0.664 | mean_pos = 0.081 (22050 samples)
[ INFO] [1583422655.052090494]:     RPE: seg 28 - mean_ori = 0.732 | mean_pos = 0.083 (20520 samples)
[ INFO] [1583422655.054294322]:     RPE: seg 35 - mean_ori = 0.793 | mean_pos = 0.090 (18960 samples)
[ INFO] [1583422655.055676035]:     RMSE: mean_ori = 0.644 | mean_pos = 0.056
[ INFO] [1583422655.056987984]:     RMSE 2D: mean_ori = 0.516 | mean_pos = 0.052
[ INFO] [1583422655.058269163]:     NEES: mean_ori = 793.646 | mean_pos = 13.095
[ INFO] [1583422656.182660653]: ======================================
[ INFO] [1583422656.183065588]: [COMP]: processing mono_ov_vio algorithm
[ INFO] [1583422656.209545279]: [TRAJ]: q_ESTtoGT = 0.000, 0.000, -0.148, 0.989 | p_ESTinGT = 0.931, 2.169,
       0.971 | s = 1.00
....
[ INFO] [1583422656.791523636]: [TRAJ]: q_ESTtoGT = 0.000, 0.000, -0.149, 0.989 | p_ESTinGT = 0.941, 2.163,
       0.974 | s = 1.00
[ INFO] [1583422656.835407991]:     ATE: mean_ori = 0.639 | mean_pos = 0.076
[ INFO] [1583422656.835433475]:     ATE: std_ori  = 0.05800 | std_pos  = 0.00430
[ INFO] [1583422656.835446222]:     ATE 2D: mean_ori = 0.514 | mean_pos = 0.070
[ INFO] [1583422656.835457020]:     ATE 2D: std_ori  = 0.07102 | std_pos  = 0.00492
[ INFO] [1583422656.838656567]:     RPE: seg 7 - mean_ori = 0.614 | mean_pos = 0.092 (25160 samples)
[ INFO] [1583422656.841540191]:     RPE: seg 14 - mean_ori = 0.634 | mean_pos = 0.092 (23470 samples)
[ INFO] [1583422656.844219466]:     RPE: seg 21 - mean_ori = 0.632 | mean_pos = 0.115 (22050 samples)
[ INFO] [1583422656.846646272]:     RPE: seg 28 - mean_ori = 0.696 | mean_pos = 0.119 (20520 samples)
[ INFO] [1583422656.848862913]:     RPE: seg 35 - mean_ori = 0.663 | mean_pos = 0.154 (18960 samples)
[ INFO] [1583422656.850321777]:     RMSE: mean_ori = 0.600 | mean_pos = 0.067
[ INFO] [1583422656.851673985]:     RMSE 2D: mean_ori = 0.479 | mean_pos = 0.060
[ INFO] [1583422656.853037942]:     NEES: mean_ori = 625.447 | mean_pos = 10.629
[ INFO] [1583422658.194763413]: ======================================
....
```

7.3.3.4 Script "error_comparison"

This compares all methods to each other on a series of datasets. For example, you run a bunch of methods on all the EurocMav datasets and then want to compare. This will do the RPE over all trajectories, and an ATE for each dataset. It will print the ATE and RPE for each method on each dataset in the console. Then following the Filter Evaluation Metrics, these are averaged over all the runs and datasets. Finally at the end it outputs a nice latex table which can be directly used in a paper. To change the RPE distances you will need to edit the code currently.

```
rosrun ov_eval error_comparison <align_mode> <folder_groundtruth> <folder_algorithms>
rosrun ov_eval error_comparison posyaw truths/ algorithms/
```

Example output:

```
[ INFO] [1583425216.054023187]: [COMP]: 2895 poses in V1_01_easy.txt => length of 58.35 meters
[ INFO] [1583425216.092355692]: [COMP]: 16702 poses in V1_02_medium.txt => length of 75.89 meters
[ INFO] [1583425216.133532429]: [COMP]: 20932 poses in V1_03_difficult.txt => length of 78.98 meters
[ INFO] [1583425216.179616651]: [COMP]: 22401 poses in V2_01_easy.txt => length of 36.50 meters
[ INFO] [1583425216.225299463]: [COMP]: 23091 poses in V2_02_medium.txt => length of 83.23 meters
[ INFO] [1583425216.225660364]: ======================================
[ INFO] [1583425223.560550101]: [COMP]: processing mono_ov_vio algorithm
[ INFO] [1583425223.560632706]: [COMP]: processing mono_ov_vio algorithm => V1_01_easy dataset
[ INFO] [1583425224.236769465]: [COMP]: processing mono_ov_vio algorithm => V1_02_medium dataset
[ INFO] [1583425224.855489521]: [COMP]: processing mono_ov_vio algorithm => V1_03_difficult dataset
[ INFO] [1583425225.659183593]: [COMP]: processing mono_ov_vio algorithm => V2_01_easy dataset
[ INFO] [1583425226.442217424]: [COMP]: processing mono_ov_vio algorithm => V2_02_medium dataset
[ INFO] [1583425227.366004330]: ======================================
....
[ INFO] [1583425261.724448413]: ============================================
[ INFO] [1583425261.724469372]: ATE LATEX TABLE
[ INFO] [1583425261.724481841]: ============================================
 & \textbf{V1\_01\_easy} & \textbf{V1\_02\_medium} & \textbf{V1\_03\_difficult}
 & \textbf{V2\_01\_easy} & \textbf{V2\_02\_medium} & \textbf{Average} \\\hline
mono\_ov\_slam & 0.699 / 0.058 & 1.675 / 0.076 & 2.542 / 0.063 & 0.773 / 0.124 & 1.538 / 0.074 & 1.445 / 0.079
        \\
mono\_ov\_vio & 0.642 / 0.076 & 1.766 / 0.096 & 2.391 / 0.344 & 1.164 / 0.121 & 1.248 / 0.106 & 1.442 / 0.148 \\
....
[ INFO] [1583425261.724647970]: ============================================
[ INFO] [1583425261.724655060]: ============================================
[ INFO] [1583425261.724661046]: RPE LATEX TABLE
[ INFO] [1583425261.724666910]: ============================================
 & \textbf{8m} & \textbf{16m} & \textbf{24m} & \textbf{32m} & \textbf{40m} & \textbf{48m} \\\hline
mono\_ov\_slam & 0.661 / 0.074 & 0.802 / 0.086 & 0.979 / 0.097 & 1.061 / 0.105 & 1.145 / 0.120 & 1.289 / 0.122
        \\
mono\_ov\_vio & 0.826 / 0.094 & 1.039 / 0.106 & 1.215 / 0.111 & 1.283 / 0.132 & 1.342 / 0.151 & 1.425 / 0.184 \\
....
[ INFO] [1583425262.514587296]: ============================================
```

7.4 Filter Timing Analysis

Installation Warning

If you plan to use the included plotting from the cpp code, you will need to make sure that you have matplotlib and python 2.7 installed. We use the to `matplotlib-cpp` to call this external library and generate the desired figures. Please see Additional Evaluation Requirements for more details on the exact install.

7.4.1 Collection

To profile the different parts of the system we record the timing information from directly inside the ov_msckf::VioManager. The file should be comma separated format, with the first column being the timing, and the last column being the total time (units are all in seconds). The middle columns should describe how much each component takes (whose names are extracted from the header of the csv file). You can use the bellow tools as long as you follow this format, and add or remove components as you see fit to the middle columns.

To evaluate the computational load (*not computation time*), we have a python script that leverages the `psutil` python package to record percent CPU and memory consumption. This can be included as an additional node in the launch file which only needs the node which you want the reported information of. This will poll the node for its percent memory, percent cpu, and total number of threads that it uses. This can be useful if you wish to compare different methods on the same platform, but doesn't make sense to use this to compare the running of the same algorithm or different algorithms *across* different hardware devices.

```
<node name="recorder_timing" pkg="ov_eval" type="pid_ros.py" output="screen">
    <param name="nodes"   type="str" value="/run_subscribe_msckf" />
    <param name="output"  type="str" value="/tmp/psutil_log.txt" />
</node>
```

It is also important to note that if the estimator has multiple nodes, you can subscribe to them all by specifying their names as a comma separated string. For example to evaluate the computation needed for `VINS-Mono` multi-node system we can do:

```
<node name="recorder_timing" pkg="ov_eval" type="pid_ros.py" output="screen">
    <param name="nodes"   type="str" value="/feature_tracker,/vins_estimator,/pose_graph" />
    <param name="output"  type="str" value="/tmp/psutil_log.txt" />
</node>
```

7.4.2 Processing & Plotting

7.4.2.1 Script "timing_flamegraph"

The flame graph script looks to recreate a `FlameGraph` of the key components of the system. While we do not trace all functions, the key "top level" function times are recorded to file to allow for insight into what is taking the majority of the computation. The file should be comma separated format, with the first column being the timing, and the last column being the total time. The middle columns should describe how much each component takes (whose names are extracted from the header of the csv file).

```
rosrun ov_eval timing_flamegraph <file_times.txt>
rosrun ov_eval timing_flamegraph timing_mono_ethV101.txt
```

Example output:

```
[TIME]: loaded 2817 timestamps from file (7 categories)!!
mean_time = 0.0037 | std = 0.0011 | 99th = 0.0063  | max = 0.0254 (tracking)
mean_time = 0.0004 | std = 0.0001 | 99th = 0.0006  | max = 0.0014 ( propagation)
mean_time = 0.0032 | std = 0.0022 | 99th = 0.0083  | max = 0.0309 (msckf update)
mean_time = 0.0034 | std = 0.0013 | 99th = 0.0063  | max = 0.0099 (slam update)
mean_time = 0.0012 | std = 0.0017 | 99th = 0.0052  | max = 0.0141 (slam delayed)
mean_time = 0.0009 | std = 0.0003 | 99th = 0.0015  | max = 0.0027 (marginalization)
mean_time = 0.0128 | std = 0.0035 | 99th = 0.0208  | max = 0.0403 (total)
```

7.4.2.2 Script "timing_comparison"

This script is use to compare the run-time of different runs of the algorithm. This take in the same file as the flame graph and is recorded in the ov_msckf::VioManager. The file should be comma separated format, with the first column being the timing, and the last column being the total time. The middle columns should describe how much each component takes (whose names are extracted from the header of the csv file).

```
rosrun ov_eval timing_comparison <file_times1.txt> ... <file_timesN.txt>
rosrun ov_eval timing_comparison timing_mono_ethV101.txt timing_stereo_ethV101.txt
```

Example output:

```
======================================
[TIME]: loading data for timing_mono
[TIME]: loaded 2817 timestamps from file (7 categories)!!
mean_time = 0.0037 | std = 0.0011 | 99th = 0.0063  | max = 0.0254 (tracking)
mean_time = 0.0004 | std = 0.0001 | 99th = 0.0006  | max = 0.0014 ( propagation)
mean_time = 0.0032 | std = 0.0022 | 99th = 0.0083  | max = 0.0309 (msckf update)
mean_time = 0.0034 | std = 0.0013 | 99th = 0.0063  | max = 0.0099 (slam update)
mean_time = 0.0012 | std = 0.0017 | 99th = 0.0052  | max = 0.0141 (slam delayed)
mean_time = 0.0009 | std = 0.0003 | 99th = 0.0015  | max = 0.0027 (marginalization)
mean_time = 0.0128 | std = 0.0035 | 99th = 0.0208  | max = 0.0403 (total)
======================================
======================================
[TIME]: loading data for timing_stereo
[TIME]: loaded 2817 timestamps from file (7 categories)!!
mean_time = 0.0077 | std = 0.0020 | 99th = 0.0124  | max = 0.0219 (tracking)
mean_time = 0.0004 | std = 0.0001 | 99th = 0.0007  | max = 0.0023 ( propagation)
mean_time = 0.0081 | std = 0.0047 | 99th = 0.0189  | max = 0.0900 (msckf update)
mean_time = 0.0063 | std = 0.0023 | 99th = 0.0117  | max = 0.0151 (slam update)
mean_time = 0.0020 | std = 0.0026 | 99th = 0.0079  | max = 0.0205 (slam delayed)
mean_time = 0.0019 | std = 0.0005 | 99th = 0.0031  | max = 0.0052 (marginalization)
mean_time = 0.0263 | std = 0.0063 | 99th = 0.0410  | max = 0.0979 (total)
======================================
```

7.4.2.3 Script "timing_percentages"

This utility allows for comparing the resources used by the algorithm on a specific platform. An example usage would be how the memory and cpu requirements increase as the state size grows or as more cameras are added. You will need to record the format using the `pid_ros.py` node (see Collection for details on how to use it). Remember that 100% cpu usage means that it takes one cpu thread to support the system, while 200% means it takes two cpu threads to support the system (see `this link` for an explanation). The folder structure needed is as follows:

```
psutil_logs/
    ov_mono/
        run1.txt
        run2.txt
        run3.txt
    ov_stereo/
        run1.txt
        run2.txt
        run3.txt
rosrun ov_eval timing_percentages <timings_folder>
rosrun ov_eval timing_percentages psutil_logs/
```

Example output:

```
======================================
[COMP]: processing ov_mono algorithm
    loaded 149 timestamps from file!!
    PREC: mean_cpu = 83.858 +- 17.130
    PREC: mean_mem = 0.266 +- 0.019
    THR: mean_threads = 12.893 +- 0.924
======================================
======================================
[COMP]: processing ov_stereo algorithm
    loaded 148 timestamps from file!!
    PREC: mean_cpu = 111.007 +- 16.519
    PREC: mean_mem = 5.139 +- 2.889
    THR: mean_threads = 12.905 +- 0.943
======================================
```

Chapter 8

Developer and Internals

- Coding Style Guide — General coding styles and conventions
- Documentation Guide — Developer guide on how documentation can be built
- ROS1 to ROS2 Bag Conversion Guide — Some notes on ROS bag conversion
- Covariance Index Internals — Description of the covariance index system
- System Profiling — Some notes on performing profiling
- Future Roadmap — Where we plan to go in the future

8.1 Coding Style Guide

8.1.1 Overview

Please note that if you have a good excuse to either break the rules or modify them, feel free to do it (and update this guide accordingly, if appropriate). Nothing is worse than rule that hurts productivity instead of improving it. In general, the main aim of this style is:

- Vertical and horizontal compression, fitting more code on a screen while keeping the code readable.
- Do not need to enforce line wrapping if clarity is impacted (i.e. Jacobians)
- Consistent indentation and formatting rules to ensure readability (4 space tabbing)

The codebase is coded in `snake_case` with accessor and getter function for most classes (there are a few exceptions). We leverage the `Doxygen` documentation system with a post-processing script from `m.css`. Please see Documentation Guide for details on how our documentation is generated. All functions should be commented both internally and externally with a focus on being as clear to a developer or user that is reading the code or documentation website.

8.1.2 Naming Conventions

We have particular naming style for our orientations and positions that should be consistent throughout the project. In general rotation matrices are the only exception of a variable that starts with a capital letter. Both position and orientation variables should contain the frame of references.

```
Eigen::Matrix<double,3,3> R_ItoC; //=> SO(3) rotation from IMU to camera frame
Eigen::Matrix<double,4,1> q_ItoC; //=> JPL quaternion rot from IMU to the camera
Eigen::Vector3d p_IinC; //=> 3d position of the IMU frame in the camera frame
Eigen::Vector3d p_FinG; //=> position of feature F in the global frame G
```

8.1.3 Print Statements

The code uses a simple print statement level logic that allows the user to enable and disable the verbosity. In general the user can specify the following (see `ov_core/src/utils/print.h` file for details):

- PrintLevel::ALL : All PRINT_XXXX will output to the console
- PrintLevel::DEBUG : "DEBUG", "INFO", "WARNING" and "ERROR" will be printed. "ALL" will be silenced
- PrintLevel::INFO : "INFO", "WARNING" and "ERROR" will be printed. "ALL" and "DEBUG" will be silenced
- PrintLevel::WARNING : "WARNING" and "ERROR" will be printed. "ALL", "DEBUG" and "INFO" will be silenced
- PrintLevel::ERROR : Only "ERROR" will be printed. All the rest are silenced
- PrintLevel::SILENT : All PRINT_XXXX will be silenced.

To use these, you can specify the following using the `printf` standard input logic. A user can also specify colors (see `ov_core/src/utils/colors.h` file for details):

```
PRINT_ALL("the value is %.2f\n", variable);
PRINT_DEBUG("the value is %.2f\n", variable);
PRINT_INFO("the value is %.2f\n", variable);
PRINT_WARNING("the value is %.2f\n", variable);
PRINT_ERROR(RED "the value is %.2f\n" RESET, variable);
```

8.2 Documentation Guide

8.2.1 Building the Latex PDF Reference Manual

1. You will need to install doxygen and texlive with all packages
   - `sudo apt-get update`
   - `sudo apt-get install doxygen texlive-full`
   - You will likely need new version of `doxygen` 1.9.4 to fix ghostscript errors
2. Go into the project folder and generate latex files
   - `cd open_vins/`
   - `doxygen`
   - This should run, and will stop if there are any latex errors
   - On my Ubuntu 20.04 this installs version "2019.20200218"
3. Generate a PDF from the latex files
   - `cd doxgen_generated/latex/`
   - `make`
   - File should be called `refman.pdf`

8.2.2 Adding a Website Page

1. Add a `doc/pagename.dox` file, copy up-to-date license header.
2. Add a `@page` definition and title to your page
3. If the page is top-level, list it in `doc/00-page-order.dox` to ensure it gets listed at a proper place
4. If the page is not top-level, list it using `@subpage` in its parent page
5. Leverage `@section` and `@subsection` to separate the page

8.2.3 Math / Latex Equations

- More details on how to format your documentation can be found here:
  - http://www.doxygen.nl/manual/formulas.html
  - https://mcss.mosra.cz/css/components/#math
  - https://mcss.mosra.cz/css/components/
- Use the inline commands for latex `\f $ <formula_here> \f $` (no space between f and $)
- Use block to have equation centered on page `\f [ <big_formula_here> \f ]` (no space between f and [])

8.2.4 Building the Documentation Site

1. Clone the m.css repository which has scripts to build
   - Ensure that it is **not** in the same folder as your open_vins
   - `git clone https://github.com/mosra/m.css`
   - This repository contains the script that will generate our documentation
2. You will need to install python3.6
   - `sudo add-apt-repository ppa:deadsnakes/ppa`
   - `sudo apt-get update`
   - `sudo apt-get install python3.6`
   - `curl https://bootstrap.pypa.io/get-pip.py | sudo python3.6`
   - `sudo -H pip3.6 install jinja2 Pygments`
   - `sudo apt install texlive-base texlive-latex-extra texlive-fonts-extra texlive-fonts-recommended`
3. To use the bibtex citing you need to have
   - bibtex executable in search path
   - perl executable in search path
   - http://www.doxygen.nl/manual/commands.html#cmdcite
4. Go into the documentation folder and build

    - `cd m.css/documentation/`
    - `python3.6 doxygen.py <path_to_Doxyfile-mcss>`

5. If you run into errors, ensure your python path is set to use the 3.6 libraries
    - `export PYTHONPATH=/usr/local/lib/python3.6/dist-packages/`
6. You might need to comment out the `enable_async=True` flag in the doxygen.py file
7. This should then build the documentation website
8. Open the html page `/open_vins/doxgen_generated/html/index.html` to view documentation

8.2.5 Custom m.css Theme

1. This is based on the `m.css docs` for custom theme
2. Files needed are in `open_vins/docs/mcss_theme/`
3. Copy the following files into the `m.css/css/` folder
    - m-theme-udel.css
    - pygments-tango.css
    - m-udel.css
4. Most settings are in the `m-theme-udel.css` file
5. To generate / compile the edited theme do:
    - `python3.6 postprocess.py m-udel.css m-documentation.css -o m-udel+documentation.com css`
    - Copy this generated file into `open_vins/docs/css/`
    - Regenerate the website and it should change the theme

8.2.6 Creating Figures

- One of the editors we use is `IPE` which is avalible of different platforms
- We use the ubuntu 16.04 version 7.1.10
- Create your figure in IPE then save it to disk (i.e. should have a `file.ipe`)
- Use the command line utility `iperender` to convert into a svg
- `iperender -svg -transparent file.ipe file.svg`

8.3 ROS1 to ROS2 Bag Conversion Guide

8.3.1 rosbags

`rosbags` is the simplest utility which does not depend on ROS installs at all. ROS bag conversion is a hard problem since you need to have both ROS1 and ROS2 dependencies. This is what was used to generate the converted ROS2 bag files for standard datasets. To do a conversion of a bag file we can do the following:

```
pip install rosbags
rosbags-convert V1_01_easy.bag
```

8.3.2 rosbag2 play

To do this conversion you will need to have both ROS1 and ROS2 installed on your system. Also ensure that you have installed all dependencies and backends required. The main rosbag2 readme has a lot of good details.

```
sudo apt-get install ros-$ROS2_DISTRO-ros2bag ros-$ROS2_DISTRO-rosbag2*
sudo apt install ros-$ROS2_DISTRO-rosbag2-bag-v2-plugins
```

From here we can do the following. This is based on this issue. You might run into issues with the .so files being corrupted (see this issue) Not sure if there is a fix besides building it from scratch yourself.

```
source_ros1
source_ros2
ros2 bag play -s rosbag_v2 V1_01_easy.bag
```

8.4 Covariance Index Internals

8.4.1 Type System

The type system that the ov_msckf leverages was inspired by graph-based optimization frameworks such as Georgia Tech Smoothing and Mapping library (GTSAM). As compared to manually knowing where in the covariance state elements are, the interaction is abstracted away from the user and is replaced by a straight forward way to access the desired elements. The ov_msckf::State is owner of these types and thus after creation one should not delete these externally.

```
class Type {
protected:
    // Current best estimate
    Eigen::MatrixXd _value;
    // Location of error state in covariance
    int _id = -1;
    // Dimension of error state
    int _size = -1;
};
```

A type is defined by its location in its covariance, its current estimate and its error state size. The current value does not have to be a vector, but could be a matrix in the case of an SO(3) rotation group type object. The error state needs to be a vector and thus a type will need to define the mapping between this error state and its manifold representation.

8.4.2 Type-based EKF Update

To show the power of this type-based indexing system, we will go through how we compute the EKF update. The specific method we will be looking at is the ov_msckf::StateHelper::EKFUpdate() which takes in the state, vector of types, Jacobian, residual, and measurement covariance. As compared to passing a Jacobian matrix that is the full size of state wide, we instead leverage this type system by passing a "small" Jacobian that has only the state elements that the measurements are a function of.

For example, if we have a global 3D SLAM feature update (implemented in ov_msckf::UpdaterSLAM) our Jacobian will only be a function of the newest clone and the feature. This means that our Jacobian is only of size 9 as compared to the size our state. In addition to the matrix containing the Jacobian elements, we need to know what order this Jacobian is in, thus we pass a vector of types which correspond to the state elements our Jacobian is a function of. Thus in our example, it would contain two types: our newest clone ov_type::PoseJPL and current landmark feature ov_type::Landmark with our Jacobian being the type size of the pose plus the type size of the landmark in width.

8.5 System Profiling

8.5.1 Profiling Processing Time

One way (besides inserting timing statements into the code) is to leverage a profiler such as `valgrind`. This tool allows for recording of the call stack of the system. To use this with a ROS node, we can do the following (based on `this` guide):

- Edit `roslaunch ov_msckf pgeneva_serial_eth.launch` launch file
- Append `launch-prefix="valgrind --tool=callgrind --callgrind-out-file=/tmp/callgrind.txt"` to your ROS node. This will cause the node to run with valgrind.
- Change the bag length to be only 10 or so seconds (since profiling is slow)

```
sudo apt install valgrind
roslaunch ov_msckf pgeneva_serial_eth.launch
```

After running the profiling program we will want to visualize it. There are some good tools for that, specifically we are using `gprof2dot` and `xdot.py`. First we will post-process it into a xdot graph format and then visualize it for inspection.

```
// install viz programs
apt-get install python3 graphviz
apt-get install gir1.2-gtk-3.0 python3-gi python3-gi-cairo graphviz
pip install gprof2dot xdot
// actually process and then viz call file
gprof2dot --format callgrind --strip /tmp/callgrind.txt --output /tmp/callgrind.xdot
xdot /tmp/callgrind.xdot
```

8.5.2 Memory Leaks

One can leverage a profiler such as `valgrind` to perform memory leak check of the codebase. Ensure you have installed the `valgrind` package (see above). We can change the node launch file as follows:

- Edit `roslaunch ov_msckf pgeneva_serial_eth.launch` launch file
- Append `launch-prefix="valgrind --tool=memcheck --leak-check=yes"` to your ROS node. This will cause the node to run with valgrind.
- Change the bag length to be only 10 or so seconds (since profiling is slow)

This `page` has some nice support material for FAQ. An example loss is shown below which was found by memcheck.

```
==5512== 1,578,860 (24 direct, 1,578,836 indirect) bytes in 1 blocks are definitely lost in loss record 6,585 of
       6,589
==5512==    at 0x4C3017F: operator new(unsigned long) (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
....
==5512==    by 0x543F868: operator[] (unordered_map.h:973)
==5512==    by 0x543F868: ov_core::TrackKLT::feed_stereo(double, cv::Mat&, cv::Mat&, unsigned long, unsigned
       long) (TrackKLT.cpp:165)
==5512==    by 0x4EF8C52: ov_msckf::VioManager::feed_measurement_stereo(double, cv::Mat&, cv::Mat&, unsigned
       long, unsigned long) (VioManager.cpp:245)
==5512==    by 0x1238A9: main (ros_serial_msckf.cpp:247)
```

8.5.3 Compiler Profiling

Here is a small guide on how to perform compiler profiling for building of the codebase. This should be used to try to minimize compile times which in general hurt developer productivity. It is recommended to read the following pages which this is a condenced form of:

- https://aras-p.info/blog/2019/01/16/time-trace-timeline-flame-chart-profiler-for-↩ Clang/
- https://aras-p.info/blog/2019/09/28/Clang-Build-Analyzer/

First we need to ensure we have a compiler that can profile the build time. Clang greater then 9 should work, but we have tested only with 11. We can get the `latest Clang` by using the follow auto-install script:

```
sudo bash -c "$(wget -O - https://apt.llvm.org/llvm.sh)"
export CC=/usr/bin/clang-11
export CXX=/usr/bin/clang++-11
```

We then need to clone the analyzer repository, which allows for summary generation.

```
git clone https://github.com/aras-p/ClangBuildAnalyzer
cd ClangBuildAnalyzer
cmake . && make
```

We can finally build our ROS package and time how long it takes. Note that we are using `catkin tools` to build here. The prefix *CBA* means to run the command in the ClangBuildAnalyzer repository clone folder. While the prefix *WS* means run in the root of your ROS workspace.

```
(WS) cd ~/workspace/
(WS) catkin clean -y && mkdir build
(CBA) ./ClangBuildAnalyzer --start ~/workspace/build/
(WS) export CC=/usr/bin/clang-11 && export CXX=/usr/bin/clang++-11
(WS) catkin build ov_msckf -DCMAKE_CXX_FLAGS="-ftime-trace"
(CBA) ./ClangBuildAnalyzer --stop ~/workspace/build/ capture_file.bin
(CBA) ./ClangBuildAnalyzer --analyze capture_file.bin > timing_results.txt
```

The `time-trace` flag should generate a bunch of .json files in your build folder. These can be opened in your chrome browser `chrome://tracing` for viewing. In general the ClangBuildAnalyzer is more useful for finding what files take long. An example output of what is generated in the timing_results.txt file is:

```
Analyzing build trace from 'capture_file.bin'...
     Time summary:
Compilation (86 times):
  Parsing (frontend):          313.9 s
  Codegen & opts (backend):    222.9 s
     Files that took longest to parse (compiler frontend):
 13139 ms: /build//ov_msckf/CMakeFiles/ov_msckf_lib.dir/src/update/UpdaterSLAM.cpp.o
 12843 ms: /build//ov_msckf/CMakeFiles/run_serial_msckf.dir/src/ros_serial_msckf.cpp.o
 ...

     Functions that took longest to compile:
  1639 ms: main (/src/open_vins/ov_eval/src/error_comparison.cpp)
  1337 ms: ov_core::BsplineSE3::get_acceleration(double, Eigen::Matrix<double, ...
       (/src/open_vins/ov_core/src/sim/BsplineSE3.cpp)
  1156 ms: ov_eval::ResultSimulation::plot_state(bool, double)
       (/src/open_vins/ov_eval/src/calc/ResultSimulation.cpp)
  ...
     Expensive headers:
 27505 ms: /src/open_vins/ov_core/src/track/TrackBase.h (included 12 times, avg 2292 ms), included via:
   TrackKLT.cpp.o TrackKLT.h  (4372 ms)
   TrackBase.cpp.o  (4297 ms)
   TrackSIM.cpp.o TrackSIM.h  (4252 ms)
   ...
```

Some key methods to reduce compile times are as follows:

- Only include headers that are required for your class
- Don't include headers in your header files `.h` that are only required in your `.cpp` source files.
- Consider `forward declarations` of methods and types
- Ensure you are using an include guard in your headers

8.6 Future Roadmap

The initial release provides the library foundation which contains a filter-based visual-inertial localization solution. This can be used in a wide range of scenarios and the type-based index system allows for others to easily add new features and develop on top of this system. Here is a list of future directions, although nothing is set in stone, that we are interested in taking:

- Creation of a secondary graph-based thread that loosely introduces loop closures (akin to the second thread of VINS-Mono and others) which should allow for drift free long-term localization.
- Large scale offline batch graph optimization which leverages the trajectory of the ov_msckf as its initial guess and then optimizes both the map and trajectory.
- Incorporate our prior work in preintegration Eckenhoff et al. [2019] into the same framework structure to allow for easy extensibility. Focus on sparsification and marginalization to allow for realtime computation.
- Leverage this sliding-window batch method to perform SFM initialization of all methods.
- Support for arbitrary Schmidt'ing of state elements allowing for modeling of their prior uncertainties but without optimizing their values online.
- More advance imu integration schemes, quantification of the integration noises to handle low frequency readings, and modeling of the imu intrinsics.

Chapter 9

Namespace Index

9.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 10

Hierarchical Index

10.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 11

Class Index

11.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 12

Namespace Documentation

12.1 ov_core Namespace Reference

Core algorithms for OpenVINS.

Classes

- class BsplineSE3

  B-Spline which performs interpolation over SE(3) manifold.
- class CamBase

  Base pinhole camera model class.
- class CamEqui

  Fisheye / equadistant model pinhole camera model class.
- struct CameraData

  Struct for a collection of camera measurements.
- class CamRadtan

  Radial-tangential / Brown–Conrady model pinhole camera model class.
- class CpiBase

  Base class for continuous preintegration integrators.
- class CpiV1

  Model 1 of continuous preintegration.
- class CpiV2

  Model 2 of continuous preintegration.
- class DatasetReader

  Helper functions to read in dataset files.
- class Feature

  Sparse feature class used to collect measurements.
- class FeatureDatabase

  Database containing features we are currently tracking.
- class FeatureHelper

  Contains some nice helper functions for features.

- class FeatureInitializer

  Class that triangulates feature.
- struct FeatureInitializerOptions

  Struct which stores all our feature initializer options.
- class Grider_FAST

  Extracts FAST features in a grid pattern.
- struct ImuData

  Struct for a single imu measurement (time, wm, am)
- class LambdaBody

  Helper class to do OpenCV parallelization.
- class Printer

  Printer for open_vins that allows for various levels of printing to be done.
- class TrackAruco

  Tracking of OpenCV Aruoc tags.
- class TrackBase

  Visual feature tracking base class.
- class TrackDescriptor

  Descriptor-based visual tracking.
- class TrackKLT

  KLT tracking of features.
- class TrackSIM

  Simulated tracker for when we already have uv measurements!
- class YamlParser

  Helper class to do OpenCV yaml parsing from both file and ROS.

Functions

- Eigen::Matrix< double, 4, 1 > rot_2_quat (const Eigen::Matrix< double, 3, 3 > &rot)

  Returns a JPL quaternion from a rotation matrix.
- Eigen::Matrix< double, 3, 3 > skew_x (const Eigen::Matrix< double, 3, 1 > &w)

  Skew-symmetric matrix from a given 3x1 vector.
- Eigen::Matrix< double, 3, 3 > quat_2_Rot (const Eigen::Matrix< double, 4, 1 > &q)

  Converts JPL quaterion to SO(3) rotation matrix.
- Eigen::Matrix< double, 4, 1 > quat_multiply (const Eigen::Matrix< double, 4, 1 > &q, const Eigen::Matrix< double, 4, 1 > &p)

  Multiply two JPL quaternions.
- Eigen::Matrix< double, 3, 1 > vee (const Eigen::Matrix< double, 3, 3 > &w_x)

  Returns vector portion of skew-symmetric.
- Eigen::Matrix< double, 3, 3 > exp_so3 (const Eigen::Matrix< double, 3, 1 > &w)

  SO(3) matrix exponential.
- Eigen::Matrix< double, 3, 1 > log_so3 (const Eigen::Matrix< double, 3, 3 > &R)

  SO(3) matrix logarithm.
- Eigen::Matrix4d exp_se3 (Eigen::Matrix< double, 6, 1 > vec)

  SE(3) matrix exponential function.
- Eigen::Matrix< double, 6, 1 > log_se3 (Eigen::Matrix4d mat)

  SE(3) matrix logarithm.

- Eigen::Matrix4d hat_se3 (const Eigen::Matrix< double, 6, 1 > &vec)

  Hat operator for R^6 -> Lie Algebra se(3)
- Eigen::Matrix4d Inv_se3 (const Eigen::Matrix4d &T)

  SE(3) matrix analytical inverse.
- Eigen::Matrix< double, 4, 1 > Inv (Eigen::Matrix< double, 4, 1 > q)

  JPL Quaternion inverse.
- Eigen::Matrix< double, 4, 4 > Omega (Eigen::Matrix< double, 3, 1 > w)

  Integrated quaternion from angular velocity.
- Eigen::Matrix< double, 4, 1 > quatnorm (Eigen::Matrix< double, 4, 1 > q_t)

  Normalizes a quaternion to make sure it is unit norm.
- Eigen::Matrix< double, 3, 3 > Jl_so3 (const Eigen::Matrix< double, 3, 1 > &w)

  Computes left Jacobian of SO(3)
- Eigen::Matrix< double, 3, 3 > Jr_so3 (const Eigen::Matrix< double, 3, 1 > &w)

  Computes right Jacobian of SO(3)
- Eigen::Matrix< double, 3, 1 > rot2rpy (const Eigen::Matrix< double, 3, 3 > &rot)

  Gets roll, pitch, yaw of argument rotation (in that order).
- Eigen::Matrix< double, 3, 3 > rot_x (double t)

  Construct rotation matrix from given roll.
- Eigen::Matrix< double, 3, 3 > rot_y (double t)

  Construct rotation matrix from given pitch.
- Eigen::Matrix< double, 3, 3 > rot_z (double t)

  Construct rotation matrix from given yaw.

12.1.1 Detailed Description

Core algorithms for OpenVINS.

This has the core algorithms that all projects within the OpenVINS ecosystem leverage. The purpose is to allow for the reuse of code that could be shared between different localization systems (i.e. msckf-based, batch-based, etc.). These algorithms are the foundation which is necessary before we can even write an estimator that can perform localization. The key components of the ov_core codebase are the following:

- 3d feature initialization (see ov_core::FeatureInitializer)
- SE(3) b-spline (see ov_core::BsplineSE3)
- KLT, descriptor, aruco, and simulation feature trackers
- Groundtruth dataset reader (see ov_core::DatasetReader)
- Quaternion and other manifold math operations
- Generic type system and their implementations (see ov_type namespace)
- Closed-form preintegration Eckenhoff et al. [2019]

Please take a look at classes that we offer for the user to leverage as each has its own documentation. If you are looking for the estimator please take a look at the ov_msckf project which leverages these algorithms. If you are looking for the different types please take a look at the ov_type namespace for the ones we have.

12.1.2 Function Documentation

12.1.2.1 exp_se3()

```
Eigen::Matrix4d ov_core::exp_se3 (
            Eigen::Matrix< double, 6, 1 > vec )  [inline]
```

SE(3) matrix exponential function.

Equation is from Ethan Eade's reference: http://ethaneade.com/lie.pdf

$$\exp([\boldsymbol{\omega},\mathbf{u}]) = \begin{bmatrix} \mathbf{R} & \mathbf{V}\mathbf{u} \\ \mathbf{0} & 1 \end{bmatrix}$$

$$\begin{aligned} \mathbf{R} &= \mathbf{I} + A\lfloor\boldsymbol{\omega}\times\rfloor + B\lfloor\boldsymbol{\omega}\times\rfloor^2 \\ \mathbf{V} &= \mathbf{I} + B\lfloor\boldsymbol{\omega}\times\rfloor + C\lfloor\boldsymbol{\omega}\times\rfloor^2 \end{aligned}$$

where we have the following definitions

$$\begin{aligned} \theta &= \sqrt{\boldsymbol{\omega}^\top\boldsymbol{\omega}} \\ A &= \sin\theta/\theta \\ B &= (1-\cos\theta)/\theta^2 \\ C &= (1-A)/\theta^2 \end{aligned}$$

Parameters

vec	6x1 in the R(6) space [omega, u]

Returns

4x4 SE(3) matrix

12.1.2.2 exp_so3()

```
Eigen::Matrix< double, 3, 3 > ov_core::exp_so3 (
            const Eigen::Matrix< double, 3, 1 > & w )  [inline]
```

SO(3) matrix exponential.

SO(3) matrix exponential mapping from the vector to SO(3) lie group. This formula ends up being the Rodrigues formula. This definition was taken from "Lie Groups for 2D and 3D Transformations" by Ethan Eade equation 15. http://ethaneade.com/lie.pdf

$$\begin{aligned} &\exp\colon \mathfrak{so}(3) \to SO(3) \\ &\exp(\mathbf{v}) = \mathbf{I} + \frac{\sin\theta}{\theta}\lfloor\mathbf{v}\times\rfloor + \frac{1-\cos\theta}{\theta^2}\lfloor\mathbf{v}\times\rfloor^2 \\ &\mathrm{where} \quad \theta^2 = \mathbf{v}^\top\mathbf{v} \end{aligned}$$

Parameters

in	*w*	3x1 vector in R(3) we will take the exponential of

Returns

SO(3) rotation matrix

12.1.2.3 hat_se3()

```
Eigen::Matrix4d ov_core::hat_se3 (
            const Eigen::Matrix< double, 6, 1 > & vec )  [inline]
```

Hat operator for R^6 -> Lie Algebra se(3)

$$\mathbf{\Omega}^\wedge = \begin{bmatrix} \lfloor\boldsymbol{\omega}\times\rfloor & \mathbf{u} \\ \mathbf{0} & 0 \end{bmatrix}$$

Parameters

vec	6x1 in the R(6) space [omega, u]

Returns

Lie algebra se(3) 4x4 matrix

12.1.2.4 Inv()

```
Eigen::Matrix< double, 4, 1 > ov_core::Inv (
            Eigen::Matrix< double, 4, 1 > q )  [inline]
```

JPL Quaternion inverse.

See equation 21 in `Indirect Kalman Filter for 3D Attitude Estimation`.

$$\bar{q}^{-1} = \begin{bmatrix} -\mathbf{q} \\ q_4 \end{bmatrix}$$

Parameters

in	*q*	quaternion we want to change

Returns

inversed quaternion

12.1.2.5 Inv_se3()

```
Eigen::Matrix4d ov_core::Inv_se3 (
            const Eigen::Matrix4d & T )  [inline]
```

SE(3) matrix analytical inverse.

It seems that using the .inverse() function is not a good way. This should be used in all cases we need the inverse instead of numerical inverse. `https://github.com/rpng/open_vins/issues/12`

$$\mathbf{T}^{-1} = \begin{bmatrix} \mathbf{R}^\top & -\mathbf{R}^\top\mathbf{p} \\ \mathbf{0} & 1 \end{bmatrix}$$

Parameters

in	*T*	SE(3) matrix

Returns

inversed SE(3) matrix

12.1.2.6 Jl_so3()

```
Eigen::Matrix< double, 3, 3 > ov_core::Jl_so3 (
            const Eigen::Matrix< double, 3, 1 > & w )  [inline]
```

Computes left Jacobian of SO(3)

The left Jacobian of SO(3) is defined equation (7.77b) in `State Estimation for Robotics` by Timothy D. Barfoot. Specifically it is the following (with $\theta = |\boldsymbol{\theta}|$ and $\mathbf{a} = \boldsymbol{\theta}/|\boldsymbol{\theta}|$):

$$J_l(\boldsymbol{\theta}) = \frac{\sin\theta}{\theta}\mathbf{I} + \Big(1 - \frac{\sin\theta}{\theta}\Big)\mathbf{a}\mathbf{a}^\top + \frac{1-\cos\theta}{\theta}\lfloor\mathbf{a}\times\rfloor$$

Parameters

w	axis-angle

Returns

The left Jacobian of SO(3)

12.1.2.7 Jr_so3()

```
Eigen::Matrix< double, 3, 3 > ov_core::Jr_so3 (
            const Eigen::Matrix< double, 3, 1 > & w )  [inline]
```

Computes right Jacobian of SO(3)

The right Jacobian of SO(3) is related to the left by Jl(-w)=Jr(w). See equation (7.87) in `State Estimation for Robotics` by Timothy D. Barfoot. See Jl_so3() for the definition of the left Jacobian of SO(3).

Parameters

w	axis-angle

Returns

The right Jacobian of SO(3)

12.1.2.8 log_se3()

```
Eigen::Matrix< double, 6, 1 > ov_core::log_se3 (
            Eigen::Matrix4d mat )  [inline]
```

SE(3) matrix logarithm.

Equation is from Ethan Eade's reference: `http://ethaneade.com/lie.pdf`

$$\begin{aligned}\boldsymbol{\omega} &= \text{skew\_offdiags}\Big(\frac{\theta}{2\sin\theta}(\mathbf{R}-\mathbf{R}^\top)\Big)\\ \mathbf{u} &= \mathbf{V}^{-1}\mathbf{t}\end{aligned}$$

where we have the following definitions

$$\theta = \arccos((\mathrm{tr}(\mathbf{R}) - 1)/2)$$
$$\mathbf{V}^{-1} = \mathbf{I} - \frac{1}{2}\lfloor \boldsymbol{\omega}\times\rfloor + \frac{1}{\theta^2}\Big(1 - \frac{A}{2B}\Big)\lfloor \boldsymbol{\omega}\times\rfloor^2$$

This function is based on the GTSAM one as the original implementation was a bit unstable. See the following:

- https://github.com/borglab/gtsam/
- https://github.com/borglab/gtsam/issues/746
- https://github.com/borglab/gtsam/pull/780

Parameters

mat	4x4 SE(3) matrix

Returns

6x1 in the R(6) space [omega, u]

12.1.2.9 log_so3()

```
Eigen::Matrix< double, 3, 1 > ov_core::log_so3 (
            const Eigen::Matrix< double, 3, 3 > & R )  [inline]
```

SO(3) matrix logarithm.

This definition was taken from "Lie Groups for 2D and 3D Transformations" by Ethan Eade equation 17 & 18. http↩://ethaneade.com/lie.pdf

$$\theta = \arccos(0.5(\mathrm{trace}(\mathbf{R}) - 1))$$
$$\lfloor \mathbf{v}\times\rfloor = \frac{\theta}{2\sin\theta}(\mathbf{R} - \mathbf{R}^\top)$$

This function is based on the GTSAM one as the original implementation was a bit unstable. See the following:

- https://github.com/borglab/gtsam/
- https://github.com/borglab/gtsam/issues/746
- https://github.com/borglab/gtsam/pull/780

Parameters

in	*R*	3x3 SO(3) rotation matrix

Returns

3x1 in the R(3) space [omegax, omegay, omegaz]

12.1.2.10 Omega()

```
Eigen::Matrix< double, 4, 4 > ov_core::Omega (
            Eigen::Matrix< double, 3, 1 > w )  [inline]
```

Integrated quaternion from angular velocity.

See equation (48) of trawny tech report `Indirect Kalman Filter for 3D Attitude Estimation`.

12.1.2.11 quat_2_Rot()

```
Eigen::Matrix< double, 3, 3 > ov_core::quat_2_Rot (
            const Eigen::Matrix< double, 4, 1 > & q )  [inline]
```

Converts JPL quaterion to SO(3) rotation matrix.

This is based on equation 62 in `Indirect Kalman Filter for 3D Attitude Estimation`:

$$\mathbf{R} = (2q_4^2 - 1)\mathbf{I}_3 - 2q_4\lfloor\mathbf{q}\times\rfloor + 2\mathbf{q}\mathbf{q}^\top$$

Parameters

in	*q*	JPL quaternion

Returns

3x3 SO(3) rotation matrix

12.1.2.12 quat_multiply()

```
Eigen::Matrix< double, 4, 1 > ov_core::quat_multiply (
            const Eigen::Matrix< double, 4, 1 > & q,
            const Eigen::Matrix< double, 4, 1 > & p )  [inline]
```

Multiply two JPL quaternions.

This is based on equation 9 in Indirect Kalman Filter for 3D Attitude Estimation. We also enforce that the quaternion is unique by having q_4 be greater than zero.

$$\bar{q} \otimes \bar{p} = \mathcal{L}(\bar{q})\bar{p} = \begin{bmatrix} q_4\mathbf{I}_3 + \lfloor\mathbf{q}\times\rfloor & \mathbf{q} \\ -\mathbf{q}^\top & q_4 \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ p_4 \end{bmatrix}$$

Parameters

in	*q*	First JPL quaternion
in	*p*	Second JPL quaternion

Returns

4x1 resulting q∗p quaternion

12.1.2.13 quatnorm()

```
Eigen::Matrix< double, 4, 1 > ov_core::quatnorm (
            Eigen::Matrix< double, 4, 1 > q_t )  [inline]
```

Normalizes a quaternion to make sure it is unit norm.

Parameters

q_t	Quaternion to normalized

Returns

Normalized quaterion

12.1.2.14 rot2rpy()

```
Eigen::Matrix< double, 3, 1 > ov_core::rot2rpy (
            const Eigen::Matrix< double, 3, 3 > & rot )  [inline]
```

Gets roll, pitch, yaw of argument rotation (in that order).

To recover the matrix: R_input = R_z(yaw)∗R_y(pitch)∗R_x(roll) If you are interested in how to compute Jacobians checkout this report: http://mars.cs.umn.edu/tr/reports/Trawny05c.pdf

Parameters

rot	Rotation matrix

Returns

roll, pitch, yaw values (in that order)

12.1.2.15 rot_2_quat()

```
Eigen::Matrix< double, 4, 1 > ov_core::rot_2_quat (
            const Eigen::Matrix< double, 3, 3 > & rot )  [inline]
```

Returns a JPL quaternion from a rotation matrix.

This is based on the equation 74 in `Indirect Kalman Filter for 3D Attitude Estimation`. In the implementation, we have 4 statements so that we avoid a division by zero and instead always divide by the largest diagonal element. This all comes from the definition of a rotation matrix, using the diagonal elements and an off-diagonal.

$$\mathbf{R}(\bar{q}) = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_3q_4) & 2(q_1q_3 - q_2q_4) \\ 2(q_1q_2 - q_3q_4) & -q_2^2 + q_2^2 - q_3^2 + q_4^2 & 2(q_2q_3 + q_1q_4) \\ 2(q_1q_3 + q_2q_4) & 2(q_2q_3 - q_1q_4) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix}$$

Parameters

`in`	*rot*	3x3 rotation matrix

Returns

4x1 quaternion

12.1.2.16 rot_x()

```
Eigen::Matrix< double, 3, 3 > ov_core::rot_x (
            double t )  [inline]
```

Construct rotation matrix from given roll.

Parameters

t	roll angle

12.1.2.17 rot_y()

```
Eigen::Matrix< double, 3, 3 > ov_core::rot_y (
            double t )  [inline]
```

Construct rotation matrix from given pitch.

Parameters

t	pitch angle

12.1.2.18 rot_z()

```
Eigen::Matrix< double, 3, 3 > ov_core::rot_z (
            double t )  [inline]
```

Construct rotation matrix from given yaw.

Parameters

t	yaw angle

12.1.2.19 skew_x()

```
Eigen::Matrix< double, 3, 3 > ov_core::skew_x (
            const Eigen::Matrix< double, 3, 1 > & w )  [inline]
```

Skew-symmetric matrix from a given 3x1 vector.

This is based on equation 6 in `Indirect Kalman Filter for 3D Attitude Estimation`:

$$\lfloor \mathbf{v}\times \rfloor = \begin{bmatrix} 0 & -v_3 & v_2 \\ v_3 & 0 & -v_1 \\ -v_2 & v_1 & 0 \end{bmatrix}$$

Parameters

in	*w*	3x1 vector to be made a skew-symmetric

Returns

3x3 skew-symmetric matrix

12.1.2.20 vee()

```
Eigen::Matrix< double, 3, 1 > ov_core::vee (
             const Eigen::Matrix< double, 3, 3 > & w_x )  [inline]
```

Returns vector portion of skew-symmetric.

See skew_x() for details.

Parameters

in	*w←_x*	skew-symmetric matrix

Returns

3x1 vector portion of skew

12.2 ov_eval Namespace Reference

Evaluation and recording utilities.

Classes

- class AlignTrajectory
  Class that given two trajectories, will align the two.
- class AlignUtils
  Helper functions for the trajectory alignment class.
- class Loader
  Has helper functions to load text files from disk and process them.
- class Recorder
  This class takes in published poses and writes them to file.
- class ResultSimulation
  A single simulation run (the full state not just pose).
- class ResultTrajectory
  A single run for a given dataset.
- struct Statistics
  Statistics object for a given set scalar time series values.

12.2.1 Detailed Description

Evaluation and recording utilities.

This project contains the key evaluation and research scripts for localization methods. These come from the necessity of trying to quantify the accuracy of the estimated trajectory while also allowing for the comparision to other methods. Please see the following documentation pages for details:

- Filter Evaluation Metrics — Definitions of different metrics for estimator accuracy.
- Filter Error Evaluation Methods — Error evaluation methods for evaluating system performance.
- Filter Timing Analysis — Timing of estimator components and complexity.

The key methods that we have included are as follows:

- Absolute trajectory error
- Relative pose error (for varying segment lengths)
- Pose to text file recorder
- Timing of system components

The absolute and relative error scripts have been implemented in C++ to allow for fast computation on multiple runs. We recommend that people look at the `rpg_trajectory_evaluation` toolbox provided by Zhang and Scaramuzza. For a background we recommend reading their `A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry` Zhang and Scaramuzza [2018] and its use in `A Benchmark Comparison of Monocular Visual-Inertial Odometry Algorithms for Flying Robots` Delmerico and Scaramuzza [2018]

12.3 ov_init Namespace Reference

State initialization code.

Classes

- class InertialInitializer

  Initializer for visual-inertial system.
- struct InertialInitializerOptions

  Struct which stores all options needed for state estimation.
- class StaticInitializer

  Initializer for a static visual-inertial system.

12.3.1 Detailed Description

State initialization code.

Right now this contains static initialization code for a visual-inertial system. It will wait for the platform to stationary, and then initialize its orientation in the gravity frame.

12.4 ov_msckf Namespace Reference

Extended Kalman Filter estimator.

Classes

- class Propagator

    Performs the state covariance and mean propagation using imu measurements.
- class ROS1Visualizer

    Helper class that will publish results onto the ROS framework.
- class ROS2Visualizer

    Helper class that will publish results onto the ROS framework.
- class RosVisualizerHelper

    Helper class that handles some common versions into and out of ROS formats.
- class Simulator

    Master simulator class that generated visual-inertial measurements.
- class State

    State of our filter.
- class StateHelper

    Helper which manipulates the State and its covariance.
- struct StateOptions

    Struct which stores all our filter options.
- class UpdaterHelper

    Class that has helper functions for our updaters.
- class UpdaterMSCKF

    Will compute the system for our sparse features and update the filter.
- struct UpdaterOptions

    Struct which stores general updater options.
- class UpdaterSLAM

    Will compute the system for our sparse SLAM features and update the filter.
- class UpdaterZeroVelocity

    Will try to detect and then update using zero velocity assumption.
- class VioManager

    Core class that manages the entire system.
- struct VioManagerOptions

    Struct which stores all options needed for state estimation.

12.4.1 Detailed Description

Extended Kalman Filter estimator.

This is an implementation of a `Multi-State Constraint Kalman Filter (MSCKF)` Mourikis and Roumeliotis [2007] which leverages inertial and visual feature information. We want to stress that this is **not** a "vanilla" implementation of the filter and instead has many more features and improvements over the original. In additional we have a modular type system which allows us to initialize and marginalizing variables out of state with ease. Please see the following documentation pages for derivation details:

- IMU Propagation Derivations — Inertial propagation derivations and details.
- Measurement Update Derivations — General state update for the different measurements.
- First-Estimate Jacobian Estimators — Background on First-Estimate Jacobians and how we use them.
- Covariance Index Internals — High level details on how the type system and covariance management works.

The key features of the system are the following:

- Sliding stochastic imu clones
- First estimate Jacobians
- Camera intrinsics and extrinsic online calibration
- Time offset between camera and imu calibration
- Standard MSCKF features with nullspace projection
- 3d SLAM feature support (with six different representations)
- Generic simulation of trajectories through and environment (see ov_msckf::Simulator)

We suggest those that are interested to first checkout the State and Propagator which should provide a nice introduction to the code. Both the slam and msckf features leverage the same Jacobian code, and thus we also recommend looking at the UpdaterHelper class for details on that.

12.5 ov_type Namespace Reference

Classes

- class IMU

  Derived Type class that implements an IMU state.
- class JPLQuat

  Derived Type class that implements JPL quaternion.
- class Landmark

  Type that implements a persistent SLAM feature.
- class LandmarkRepresentation

  Class has useful feature representation types.
- class PoseJPL

  Derived Type class that implements a 6 d.o.f pose.
- class Type

  Base class for estimated variables.
- class Vec

  Derived Type class that implements vector variables.

12.5.1 Detailed Description

Types leveraged by the EKF system for covariance management. These types store where they are in the covariance along with their current estimate. Each also has an update function that takes a vector delta and updates their manifold representation. Please see each type for details on what they represent, but their names should be straightforward. See Covariance Index Internals for high level details on how the type system and covariance management works. Each type is described by the following:

```
class Type {
protected:
  // Current best estimate
  Eigen::MatrixXd _value;
  // Location of error state in covariance
  int _id = -1;
  // Dimension of error state
  int _size = -1;
  // Update eq. taking vector to their rep.
  void update(const Eigen::VectorXd dx);
};
```

Chapter 13

Class Documentation

13.1 ov_eval::AlignTrajectory Class Reference

Class that given two trajectories, will align the two.

```
#include <AlignTrajectory.h>
```

Static Public Member Functions

- static void align_trajectory (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t, double &s, std::string method, int n_aligned=-1)

  Align estimate to GT using a desired method using a set of initial poses.

Static Protected Member Functions

- static void align_posyaw_single (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t)

  Align estimate to GT using only position and yaw (for gravity aligned trajectories) using only the first poses.
- static void align_posyaw (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t, int n_aligned=-1)

  Align estimate to GT using only position and yaw (for gravity aligned trajectories) using a set of initial poses.
- static void align_se3_single (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t)

  Align estimate to GT using a full SE(3) transform using only the first poses.
- static void align_se3 (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen::↩ Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t, int n_aligned=-1)

  Align estimate to GT using a full SE(3) transform using a set of initial poses.
- static void align_sim3 (const std::vector< Eigen::Matrix< double, 7, 1 > > &traj_es, const std::vector< Eigen↩ ::Matrix< double, 7, 1 > > &traj_gt, Eigen::Matrix3d &R, Eigen::Vector3d &t, double &s, int n_aligned=-1)

  Align estimate to GT using a full SIM(3) transform using a set of initial poses.

13.1.1 Detailed Description

Class that given two trajectories, will align the two.

Given two trajectories that have already been time synchronized we will compute the alignment transform between the two. We can do this using different alignment methods such as full SE(3) transform, just postiion and yaw, or SIM(3). These scripts are based on the rpg_trajectory_evaluation toolkit by Zhang and Scaramuzza. Please take a look at their 2018 IROS paper on these methods.

13.1.2 Member Function Documentation

13.1.2.1 align_posyaw()

```
void AlignTrajectory::align_posyaw (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t,
            int n_aligned = -1 )  [static], [protected]
```

Align estimate to GT using only position and yaw (for gravity aligned trajectories) using a set of initial poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed
n_aligned	Number of poses to use for alignment (-1 will use all)

13.1.2.2 align_posyaw_single()

```
void AlignTrajectory::align_posyaw_single (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t )  [static], [protected]
```

Align estimate to GT using only position and yaw (for gravity aligned trajectories) using only the first poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed

13.1.2.3 align_se3()

```
void AlignTrajectory::align_se3 (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t,
            int n_aligned = -1 )  [static], [protected]
```

Align estimate to GT using a full SE(3) transform using a set of initial poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed
n_aligned	Number of poses to use for alignment (-1 will use all)

13.1.2.4 align_se3_single()

```
void AlignTrajectory::align_se3_single (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t )  [static], [protected]
```

Align estimate to GT using a full SE(3) transform using only the first poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed

13.1.2.5 align_sim3()

```
void AlignTrajectory::align_sim3 (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t,
            double & s,
            int n_aligned = -1 )  [static], [protected]
```

Align estimate to GT using a full SIM(3) transform using a set of initial poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed
s	scale from estimate to GT frame that will be computed
n_aligned	Number of poses to use for alignment (-1 will use all)

13.1.2.6 align_trajectory()

```
void AlignTrajectory::align_trajectory (
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_es,
            const std::vector< Eigen::Matrix< double, 7, 1 > > & traj_gt,
            Eigen::Matrix3d & R,
            Eigen::Vector3d & t,
            double & s,
            std::string method,
            int n_aligned = -1 )  [static]
```

Align estimate to GT using a desired method using a set of initial poses.

Parameters

traj_es	Estimated trajectory values in estimate frame [pos,quat]
traj_gt	Groundtruth trjaectory in groundtruth frame [pos,quat]
R	Rotation from estimate to GT frame that will be computed
t	translation from estimate to GT frame that will be computed
s	scale from estimate to GT frame that will be computed
method	Method used for alignment
n_aligned	Number of poses to use for alignment (-1 will use all)

13.2 ov_eval::AlignUtils Class Reference

Helper functions for the trajectory alignment class.

`#include <AlignUtils.h>`

Static Public Member Functions

- static double get_best_yaw (const Eigen::Matrix< double, 3, 3 > &C)

    Gets best yaw from Frobenius problem. Equation (17)-(18) in `Zhang et al. 2018 IROS` *paper.*
- static Eigen::Matrix< double, 3, 1 > get_mean (const std::vector< Eigen::Matrix< double, 3, 1 > > &data)

    Gets mean of the vector of data.
- static void align_umeyama (const std::vector< Eigen::Matrix< double, 3, 1 > > &data, const std::vector< Eigen::Matrix< double, 3, 1 > > &model, Eigen::Matrix< double, 3, 3 > &R, Eigen::Matrix< double, 3, 1 > &t, double &s, bool known_scale, bool yaw_only)

    Given a set of points in a model frame and a set of points in a data frame, finds best transform between frames (subject to constraints).
- static void perform_association (double offset, double max_difference, std::vector< double > &est_times, std↩ ::vector< double > &gt_times, std::vector< Eigen::Matrix< double, 7, 1 > > &est_poses, std::vector< Eigen↩ ::Matrix< double, 7, 1 > > &gt_poses)

    Will intersect our loaded data so that we have common timestamps.
- static void perform_association (double offset, double max_difference, std::vector< double > &est_times, std↩ ::vector< double > &gt_times, std::vector< Eigen::Matrix< double, 7, 1 > > &est_poses, std::vector< Eigen::↩ Matrix< double, 7, 1 > > &gt_poses, std::vector< Eigen::Matrix3d > &est_covori, std::vector< Eigen::Matrix3d > &est_covpos, std::vector< Eigen::Matrix3d > &gt_covori, std::vector< Eigen::Matrix3d > &gt_covpos)

    Will intersect our loaded data so that we have common timestamps.

13.2.1 Detailed Description

Helper functions for the trajectory alignment class.

The key function is an implementation of Umeyama's `Least-squares estimation of transformation parameters between two point patterns`. This is what allows us to find the transform between the two trajectories without worrying about singularities for the absolute trajectory error.

13.2.2 Member Function Documentation

13.2.2.1 align_umeyama()

```
void AlignUtils::align_umeyama (
             const std::vector< Eigen::Matrix< double, 3, 1 > > & data,
             const std::vector< Eigen::Matrix< double, 3, 1 > > & model,
             Eigen::Matrix< double, 3, 3 > & R,
             Eigen::Matrix< double, 3, 1 > & t,
             double & s,
             bool known_scale,
             bool yaw_only ) [static]
```

Given a set of points in a model frame and a set of points in a data frame, finds best transform between frames (subject to constraints).

Parameters

data	Vector of points in data frame (i.e., estimates)
model	Vector of points in model frame (i.e., gt)
R	Output rotation from data frame to model frame
t	Output translation from data frame to model frame
s	Output scale from data frame to model frame
known_scale	Whether to fix scale
yaw_only	Whether to only use yaw orientation (such as when frames are already gravity aligned)

13.2.2.2 get_best_yaw()

```
static double ov_eval::AlignUtils::get_best_yaw (
            const Eigen::Matrix< double, 3, 3 > & C )  [inline], [static]
```

Gets best yaw from Frobenius problem. Equation (17)-(18) in `Zhang et al. 2018 IROS` paper.

Parameters

C	Data matrix

13.2.2.3 get_mean()

```
static Eigen::Matrix< double, 3, 1 > ov_eval::AlignUtils::get_mean (
            const std::vector< Eigen::Matrix< double, 3, 1 > > & data )  [inline], [static]
```

Gets mean of the vector of data.

Parameters

data	Vector of data

Returns

Mean value

13.2.2.4 perform_association() [1/2]

```
void AlignUtils::perform_association (
            double offset,
            double max_difference,
            std::vector< double > & est_times,
            std::vector< double > & gt_times,
            std::vector< Eigen::Matrix< double, 7, 1 > > & est_poses,
            std::vector< Eigen::Matrix< double, 7, 1 > > & gt_poses )  [static]
```

Will intersect our loaded data so that we have common timestamps.

Parameters

offset	Time offset to append to our estimate
max_difference	Biggest allowed difference between matched timesteps

13.2.2.5 perform_association() [2/2]

```
void AlignUtils::perform_association (
            double offset,
            double max_difference,
            std::vector< double > & est_times,
            std::vector< double > & gt_times,
            std::vector< Eigen::Matrix< double, 7, 1 > > & est_poses,
            std::vector< Eigen::Matrix< double, 7, 1 > > & gt_poses,
            std::vector< Eigen::Matrix3d > & est_covori,
            std::vector< Eigen::Matrix3d > & est_covpos,
            std::vector< Eigen::Matrix3d > & gt_covori,
            std::vector< Eigen::Matrix3d > & gt_covpos )  [static]
```

Will intersect our loaded data so that we have common timestamps.

Parameters

offset	Time offset to append to our estimate
max_difference	Biggest allowed difference between matched timesteps

13.3 ov_core::BsplineSE3 Class Reference

B-Spline which performs interpolation over SE(3) manifold.

```
#include <BsplineSE3.h>
```

Public Member Functions

- **BsplineSE3** ()
  Default constructor.
- void feed_trajectory (std::vector< Eigen::VectorXd > traj_points)
  Will feed in a series of poses that we will then convert into control points.
- bool get_pose (double timestamp, Eigen::Matrix3d &R_GtoI, Eigen::Vector3d &p_IinG)
  Gets the orientation and position at a given timestamp.
- bool get_velocity (double timestamp, Eigen::Matrix3d &R_GtoI, Eigen::Vector3d &p_IinG, Eigen::Vector3d &w_↩ IinI, Eigen::Vector3d &v_IinG)
  Gets the angular and linear velocity at a given timestamp.
- bool get_acceleration (double timestamp, Eigen::Matrix3d &R_GtoI, Eigen::Vector3d &p_IinG, Eigen::Vector3d &w_IinI, Eigen::Vector3d &v_IinG, Eigen::Vector3d &alpha_IinI, Eigen::Vector3d &a_IinG)
  Gets the angular and linear acceleration at a given timestamp.
- double **get_start_time** ()
  Returns the simulation start time that we should start simulating from.

Protected Types

- typedef std::map< double, Eigen::Matrix4d, std::less< double >, Eigen::aligned_allocator< std::pair< const double, Eigen::Matrix4d > > > **AlignedEigenMat4d**
  Type defintion of our aligned eigen4d matrix: `https://eigen.tuxfamily.org/dox/group__TopicStl↩ Containers.html`.

Static Protected Member Functions

- static bool find_bounding_poses (const double timestamp, const AlignedEigenMat4d &poses, double &t0, Eigen↩ ::Matrix4d &pose0, double &t1, Eigen::Matrix4d &pose1)
  Will find the two bounding poses for a given timestamp.
- static bool find_bounding_control_points (const double timestamp, const AlignedEigenMat4d &poses, double &t0, Eigen::Matrix4d &pose0, double &t1, Eigen::Matrix4d &pose1, double &t2, Eigen::Matrix4d &pose2, double &t3, Eigen::Matrix4d &pose3)
  Will find two older poses and two newer poses for the current timestamp.

Protected Attributes

- double **dt**
  Uniform sampling time for our control points.
- double **timestamp_start**
  Start time of the system.
- AlignedEigenMat4d **control_points**
  Our control SE3 control poses (R_ItoG, p_IinG)

13.3.1 Detailed Description

B-Spline which performs interpolation over SE(3) manifold.

This class implements the b-spline functionality that allows for interpolation over the $\mathbb{SE}(3)$ manifold. This is based off of the derivations from `Continuous-Time Visual-Inertial Odometry for Event Cameras` and `A Spline-Based Trajectory Representation for Sensor Fusion and Rolling Shutter Cameras` with some additional derivations being available in `these notes`. The use of b-splines for $\mathbb{SE}(3)$ interpolation has the following properties:

1. Local control, allowing the system to function online as well as in batch
2. C^2-continuity to enable inertial predictions and calculations
3. Good approximation of minimal torque trajectories
4. A parameterization of rigid-body motion devoid of singularities

The key idea is to convert a set of trajectory points into a continuous-time *uniform cubic cumulative* b-spline. As compared to standard b-spline representations, the cumulative form ensures local continuity which is needed for on-manifold interpolation. We leverage the cubic b-spline to ensure C^2-continuity to ensure that we can calculate accelerations at any point along the trajectory. The general equations are the following

$$
\begin{aligned}
{}^{w}_{s}\mathbf{T}(u(t)) &= {}^{w}_{i-1}\mathbf{T}\ \mathbf{A}_0\ \mathbf{A}_1\ \mathbf{A}_2 \\
{}^{w}_{s}\dot{\mathbf{T}}(u(t)) &= {}^{w}_{i-1}\mathbf{T}\Big(\dot{\mathbf{A}}_0\ \mathbf{A}_1\ \mathbf{A}_2 + \mathbf{A}_0\ \dot{\mathbf{A}}_1\ \mathbf{A}_2 + \mathbf{A}_0\ \mathbf{A}_1\ \dot{\mathbf{A}}_2\Big) \\
{}^{w}_{s}\ddot{\mathbf{T}}(u(t)) &= {}^{w}_{i-1}\mathbf{T}\Big(\ddot{\mathbf{A}}_0\ \mathbf{A}_1\ \mathbf{A}_2 + \mathbf{A}_0\ \ddot{\mathbf{A}}_1\ \mathbf{A}_2 + \mathbf{A}_0\ \mathbf{A}_1\ \ddot{\mathbf{A}}_2 \\
&\qquad + 2\dot{\mathbf{A}}_0\dot{\mathbf{A}}_1\mathbf{A}_2 + 2\mathbf{A}_0\dot{\mathbf{A}}_1\dot{\mathbf{A}}_2 + 2\dot{\mathbf{A}}_0\mathbf{A}_1\dot{\mathbf{A}}_2\Big)
\end{aligned}
$$

$$
\begin{aligned}
{}^{i-1}_{i}\mathbf{\Omega} &= \log\big({}^{w}_{i-1}\mathbf{T}^{-1}\ {}^{w}_{i}\mathbf{T}\big) \\
\mathbf{A}_j &= \exp\Big(B_j(u(t))\ {}^{i-1+j}_{i+j}\mathbf{\Omega}\Big) \\
\dot{\mathbf{A}}_j &= \dot{B}_j(u(t))\ {}^{i-1+j}_{i+j}\mathbf{\Omega}^\wedge\ \mathbf{A}_j \\
\ddot{\mathbf{A}}_j &= \dot{B}_j(u(t))\ {}^{i-1+j}_{i+j}\mathbf{\Omega}^\wedge\ \dot{\mathbf{A}}_j + \ddot{B}_j(u(t))\ {}^{i-1+j}_{i+j}\mathbf{\Omega}^\wedge\ \mathbf{A}_j
\end{aligned}
$$

$$
\begin{aligned}
B_0(u(t)) &= \frac{1}{3!}\ (5 + 3u - 3u^2 + u^3) \\
B_1(u(t)) &= \frac{1}{3!}\ (1 + 3u + 3u^2 - 2u^3) \\
B_2(u(t)) &= \frac{1}{3!}\ (u^3)
\end{aligned}
$$

$$
\begin{aligned}
\dot{B}_0(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t}\ (3 - 6u + 3u^2) \\
\dot{B}_1(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t}\ (3 + 6u - 6u^2) \\
\dot{B}_2(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t}\ (3u^2)
\end{aligned}
$$

$$
\begin{aligned}
\ddot{B}_0(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t^2}\ (-6 + 6u) \\
\ddot{B}_1(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t^2}\ (6 - 12u) \\
\ddot{B}_2(u(t)) &= \frac{1}{3!}\ \frac{1}{\Delta t^2}\ (6u)
\end{aligned}
$$

where $u(t_s) = (t_s - t_i)/\Delta t = (t_s - t_i)/(t_{i+1} - t_i)$ is used for all values of *u*. Note that one needs to ensure that they use the SE(3) matrix expodential, logorithm, and hat operation for all above equations. The indexes correspond to the the two poses that are older and two poses that are newer then the current time we want to get (i.e. i-1 and i are less than s, while i+1 and i+2 are both greater than time s). Some additional derivations are available in `these notes`.

13.3.2 Member Function Documentation

13.3.2.1 feed_trajectory()

```
void BsplineSE3::feed_trajectory (
            std::vector< Eigen::VectorXd > traj_points )
```

Will feed in a series of poses that we will then convert into control points.

Our control points need to be uniformly spaced over the trajectory, thus given a trajectory we will uniformly sample based on the average spacing between the pose points specified.

Parameters

traj_points	Trajectory poses that we will convert into control points (timestamp(s), q_GtoI, p_IinG)

13.3.2.2 find_bounding_control_points()

```
bool BsplineSE3::find_bounding_control_points (
             const double timestamp,
             const AlignedEigenMat4d & poses,
             double & t0,
             Eigen::Matrix4d & pose0,
             double & t1,
             Eigen::Matrix4d & pose1,
             double & t2,
             Eigen::Matrix4d & pose2,
             double & t3,
             Eigen::Matrix4d & pose3 )  [static], [protected]
```

Will find two older poses and two newer poses for the current timestamp.

Parameters

timestamp	Desired timestamp we want to get four bounding poses of
poses	Map of poses and timestamps
t0	Timestamp of the first pose
pose0	SE(3) pose of the first pose
t1	Timestamp of the second pose
pose1	SE(3) pose of the second pose
t2	Timestamp of the third pose
pose2	SE(3) pose of the third pose
t3	Timestamp of the fourth pose
pose3	SE(3) pose of the fourth pose

Returns

False if we are unable to find bounding poses

13.3.2.3 find_bounding_poses()

```
bool BsplineSE3::find_bounding_poses (
             const double timestamp,
             const AlignedEigenMat4d & poses,
             double & t0,
             Eigen::Matrix4d & pose0,
             double & t1,
             Eigen::Matrix4d & pose1 )  [static], [protected]
```

Will find the two bounding poses for a given timestamp.

This will loop through the passed map of poses and find two bounding poses. If there are no bounding poses then this will return false.

Parameters

timestamp	Desired timestamp we want to get two bounding poses of
poses	Map of poses and timestamps
t0	Timestamp of the first pose
pose0	SE(3) pose of the first pose
t1	Timestamp of the second pose
pose1	SE(3) pose of the second pose

Returns

False if we are unable to find bounding poses

13.3.2.4 get_acceleration()

```
bool BsplineSE3::get_acceleration (
                  double timestamp,
                  Eigen::Matrix3d & R_GtoI,
                  Eigen::Vector3d & p_IinG,
                  Eigen::Vector3d & w_IinI,
                  Eigen::Vector3d & v_IinG,
                  Eigen::Vector3d & alpha_IinI,
                  Eigen::Vector3d & a_IinG )
```

Gets the angular and linear acceleration at a given timestamp.

Parameters

timestamp	Desired time to get the pose at
R_GtoI	SO(3) orientation of the pose in the global frame
p_IinG	Position of the pose in the global
w_IinI	Angular velocity in the inertial frame
v_IinG	Linear velocity in the global frame
alpha_IinI	Angular acceleration in the inertial frame
a_IinG	Linear acceleration in the global frame

Returns

False if we can't find it

13.3.2.5 get_pose()

```
bool BsplineSE3::get_pose (
             double timestamp,
             Eigen::Matrix3d & R_GtoI,
             Eigen::Vector3d & p_IinG )
```

Gets the orientation and position at a given timestamp.

Parameters

timestamp	Desired time to get the pose at
R_GtoI	SO(3) orientation of the pose in the global frame
p_IinG	Position of the pose in the global

Returns

False if we can't find it

13.3.2.6 get_velocity()

```
bool BsplineSE3::get_velocity (
             double timestamp,
             Eigen::Matrix3d & R_GtoI,
             Eigen::Vector3d & p_IinG,
             Eigen::Vector3d & w_IinI,
             Eigen::Vector3d & v_IinG )
```

Gets the angular and linear velocity at a given timestamp.

Parameters

timestamp	Desired time to get the pose at
R_GtoI	SO(3) orientation of the pose in the global frame
p_IinG	Position of the pose in the global
w_IinI	Angular velocity in the inertial frame
v_IinG	Linear velocity in the global frame

Returns

False if we can't find it

13.4 ov_core::CamBase Class Reference

Base pinhole camera model class.

```
#include <CamBase.h>
```

Public Member Functions

- CamBase (int width, int height)

  Default constructor.
- virtual void set_value (const Eigen::MatrixXd &calib)

  This will set and update the camera calibration values. This should be called on startup for each camera and after update!
- virtual Eigen::Vector2f undistort_f (const Eigen::Vector2f &uv_dist)=0

  Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.
- Eigen::Vector2d undistort_d (const Eigen::Vector2d &uv_dist)

  Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.
- cv::Point2f undistort_cv (const cv::Point2f &uv_dist)

  Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.
- virtual Eigen::Vector2f distort_f (const Eigen::Vector2f &uv_norm)=0

  Given a normalized uv coordinate this will distort it to the raw image plane.
- Eigen::Vector2d distort_d (const Eigen::Vector2d &uv_norm)

  Given a normalized uv coordinate this will distort it to the raw image plane.
- cv::Point2f distort_cv (const cv::Point2f &uv_norm)

  Given a normalized uv coordinate this will distort it to the raw image plane.
- virtual void compute_distort_jacobian (const Eigen::Vector2d &uv_norm, Eigen::MatrixXd &H_dz_dzn, Eigen::↩ MatrixXd &H_dz_dzeta)=0

  Computes the derivative of raw distorted to normalized coordinate.
- Eigen::MatrixXd **get_value** ()

  Gets the complete intrinsic vector.
- cv::Matx33d **get_K** ()

  Gets the camera matrix.
- cv::Vec4d **get_D** ()

  Gets the camera distortion.
- int **w** ()

  Gets the width of the camera images.
- int **h** ()

  Gets the height of the camera images.

Protected Attributes

- Eigen::MatrixXd **camera_values**

  Raw set of camera intrinic values (f_x & f_y & c_x & c_y & k_1 & k_2 & k_3 & k_4)
- cv::Matx33d **camera_k_OPENCV**

  Camera intrinsics in OpenCV format.
- cv::Vec4d **camera_d_OPENCV**

  Camera distortion in OpenCV format.
- int **_width**

  Width of the camera (raw pixels)
- int **_height**

  Height of the camera (raw pixels)

13.4.1 Detailed Description

Base pinhole camera model class.

This is the base class for all our camera models. All these models are pinhole cameras, thus just have standard reprojection logic.

See each base class for detailed examples on each model:

- ov_core::CamEqui
- ov_core::CamRadtan

13.4.2 Constructor & Destructor Documentation

13.4.2.1 CamBase()

```
ov_core::CamBase::CamBase (
                int width,
                int height )  [inline]
```

Default constructor.

Parameters

width	Width of the camera (raw pixels)
height	Height of the camera (raw pixels)

13.4.3 Member Function Documentation

13.4.3.1 compute_distort_jacobian()

```
virtual void ov_core::CamBase::compute_distort_jacobian (
             const Eigen::Vector2d & uv_norm,
             Eigen::MatrixXd & H_dz_dzn,
             Eigen::MatrixXd & H_dz_dzeta )  [pure virtual]
```

Computes the derivative of raw distorted to normalized coordinate.

Parameters

uv_norm	Normalized coordinates we wish to distort
H_dz_dzn	Derivative of measurement z in respect to normalized
H_dz_dzeta	Derivative of measurement z in respect to intrinsic parameters

Implemented in ov_core::CamEqui, and ov_core::CamRadtan.

13.4.3.2 distort_cv()

```
cv::Point2f ov_core::CamBase::distort_cv (
             const cv::Point2f & uv_norm )  [inline]
```

Given a normalized uv coordinate this will distort it to the raw image plane.

Parameters

uv_norm	Normalized coordinates we wish to distort

Returns

2d vector of raw uv coordinate

13.4.3.3 distort_d()

```
Eigen::Vector2d ov_core::CamBase::distort_d (
             const Eigen::Vector2d & uv_norm )  [inline]
```

Given a normalized uv coordinate this will distort it to the raw image plane.

Parameters

uv_norm	Normalized coordinates we wish to distort

Returns

2d vector of raw uv coordinate

13.4.3.4 distort_f()

```
virtual Eigen::Vector2f ov_core::CamBase::distort_f (
             const Eigen::Vector2f & uv_norm )  [pure virtual]
```

Given a normalized uv coordinate this will distort it to the raw image plane.

Parameters

uv_norm	Normalized coordinates we wish to distort

Returns

2d vector of raw uv coordinate

Implemented in ov_core::CamEqui, and ov_core::CamRadtan.

13.4.3.5 set_value()

```
virtual void ov_core::CamBase::set_value (
             const Eigen::MatrixXd & calib )  [inline], [virtual]
```

This will set and update the camera calibration values. This should be called on startup for each camera and after update!

Parameters

calib	Camera calibration information (f_x & f_y & c_x & c_y & k_1 & k_2 & k_3 & k_4)

13.4.3.6 undistort_cv()

```
cv::Point2f ov_core::CamBase::undistort_cv (
             const cv::Point2f & uv_dist )  [inline]
```

Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.

Parameters

uv_dist	Raw uv coordinate we wish to undistort

Returns

2d vector of normalized coordinates

13.4.3.7 undistort_d()

```
Eigen::Vector2d ov_core::CamBase::undistort_d (
             const Eigen::Vector2d & uv_dist )  [inline]
```

Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.

Parameters

uv_dist	Raw uv coordinate we wish to undistort

Returns

2d vector of normalized coordinates

13.4.3.8 undistort_f()

```
virtual Eigen::Vector2f ov_core::CamBase::undistort_f (
             const Eigen::Vector2f & uv_dist )  [pure virtual]
```

Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.

Parameters

uv_dist	Raw uv coordinate we wish to undistort

Returns

2d vector of normalized coordinates

Implemented in ov_core::CamEqui, and ov_core::CamRadtan.

13.5 ov_core::CamEqui Class Reference

Fisheye / equadistant model pinhole camera model class.

```
#include <CamEqui.h>
```

Public Member Functions

- CamEqui (int width, int height)

  Default constructor.
- Eigen::Vector2f undistort_f (const Eigen::Vector2f &uv_dist) override

  Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.
- Eigen::Vector2f distort_f (const Eigen::Vector2f &uv_norm) override

  Given a normalized uv coordinate this will distort it to the raw image plane.
- void compute_distort_jacobian (const Eigen::Vector2d &uv_norm, Eigen::MatrixXd &H_dz_dzn, Eigen::MatrixXd &H_dz_dzeta) override

  Computes the derivative of raw distorted to normalized coordinate.

Additional Inherited Members

13.5.1 Detailed Description

Fisheye / equadistant model pinhole camera model class.

As fisheye or wide-angle lenses are widely used in practice, we here provide mathematical derivations of such distortion model as in `OpenCV fisheye`.

$$\begin{bmatrix} u \\ v \end{bmatrix} := \mathbf{z}_k = \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta}) = \begin{bmatrix} f_x * x + c_x \\ f_y * y + c_y \end{bmatrix}$$

$$\begin{aligned} \text{where}\ \ x &= \frac{x_n}{r} * \theta_d \\ y &= \frac{y_n}{r} * \theta_d \\ \theta_d &= \theta(1 + k_1\theta^2 + k_2\theta^4 + k_3\theta^6 + k_4\theta^8) \\ r^2 &= x_n^2 + y_n^2 \\ \theta &= atan(r) \end{aligned}$$

where $\mathbf{z}_{n,k} = [x_n \; y_n]^\top$ are the normalized coordinates of the 3D feature and u and v are the distorted image coordinates on the image plane. Clearly, the following distortion intrinsic parameters are used in the above model:

$$\boldsymbol{\zeta} = \begin{bmatrix} f_x & f_y & c_x & c_y & k_1 & k_2 & k_3 & k_4 \end{bmatrix}^\top$$

In analogy to the previous radial distortion (see ov_core::CamRadtan) case, the following Jacobian for these parameters is needed for intrinsic calibration:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \boldsymbol{\zeta}} = \begin{bmatrix} x_n & 0 & 1 & 0 & f_x * (\frac{x_n}{r}\theta^3) & f_x * (\frac{x_n}{r}\theta^5) & f_x * (\frac{x_n}{r}\theta^7) & f_x * (\frac{x_n}{r}\theta^9) \\ 0 & y_n & 0 & 1 & f_y * (\frac{y_n}{r}\theta^3) & f_y * (\frac{y_n}{r}\theta^5) & f_y * (\frac{y_n}{r}\theta^7) & f_y * (\frac{y_n}{r}\theta^9) \end{bmatrix}$$

Similarly, with the chain rule of differentiation, we can compute the following Jacobian with respect to the normalized coordinates:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} = \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial x_n y_n} + \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial r}\frac{\partial r}{\partial x_n y_n} + \frac{\partial uv}{\partial xy}\frac{\partial xy}{\partial \theta_d}\frac{\partial \theta_d}{\partial \theta}\frac{\partial \theta}{\partial r}\frac{\partial r}{\partial x_n y_n}$$

$$\begin{aligned} \text{where} \quad \frac{\partial uv}{\partial xy} &= \begin{bmatrix} f_x & 0 \\ 0 & f_y \end{bmatrix} \\ \frac{\partial xy}{\partial x_n y_n} &= \begin{bmatrix} \theta_d/r & 0 \\ 0 & \theta_d/r \end{bmatrix} \\ \frac{\partial xy}{\partial r} &= \begin{bmatrix} -\frac{x_n}{r^2}\theta_d \\ -\frac{y_n}{r^2}\theta_d \end{bmatrix} \\ \frac{\partial r}{\partial x_n y_n} &= \begin{bmatrix} \frac{x_n}{r} & \frac{y_n}{r} \end{bmatrix} \\ \frac{\partial xy}{\partial \theta_d} &= \begin{bmatrix} \frac{x_n}{r} \\ \frac{y_n}{r} \end{bmatrix} \\ \frac{\partial \theta_d}{\partial \theta} &= \begin{bmatrix} 1 + 3k_1\theta^2 + 5k_2\theta^4 + 7k_3\theta^6 + 9k_4\theta^8 \end{bmatrix} \\ \frac{\partial \theta}{\partial r} &= \begin{bmatrix} \frac{1}{r^2+1} \end{bmatrix} \end{aligned}$$

To equate this to one of Kalibr's models, this is what you would use for `pinhole-equi`.

13.5.2 Constructor & Destructor Documentation

13.5.2.1 CamEqui()

```
ov_core::CamEqui::CamEqui (
              int width,
              int height )  [inline]
```

Default constructor.

Parameters

width	Width of the camera (raw pixels)
height	Height of the camera (raw pixels)

13.5.3 Member Function Documentation

13.5.3.1 compute_distort_jacobian()

```
void ov_core::CamEqui::compute_distort_jacobian (
             const Eigen::Vector2d & uv_norm,
             Eigen::MatrixXd & H_dz_dzn,
             Eigen::MatrixXd & H_dz_dzeta )  [inline], [override], [virtual]
```

Computes the derivative of raw distorted to normalized coordinate.

Parameters

uv_norm	Normalized coordinates we wish to distort
H_dz_dzn	Derivative of measurement z in respect to normalized
H_dz_dzeta	Derivative of measurement z in respect to intrinic parameters

Implements ov_core::CamBase.

13.5.3.2 distort_f()

```
Eigen::Vector2f ov_core::CamEqui::distort_f (
             const Eigen::Vector2f & uv_norm )  [inline], [override], [virtual]
```

Given a normalized uv coordinate this will distort it to the raw image plane.

Parameters

uv_norm	Normalized coordinates we wish to distort

Returns

2d vector of raw uv coordinate

Implements ov_core::CamBase.

13.5.3.3 undistort_f()

```
Eigen::Vector2f ov_core::CamEqui::undistort_f (
            const Eigen::Vector2f & uv_dist )  [inline], [override], [virtual]
```

Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.

Parameters

uv_dist	Raw uv coordinate we wish to undistort

Returns

2d vector of normalized coordinates

Implements ov_core::CamBase.

13.6 ov_core::CameraData Struct Reference

Struct for a collection of camera measurements.

`#include <sensor_data.h>`

Public Member Functions

- bool **operator<** (const CameraData &other) const

  Sort function to allow for using of STL containers.

Public Attributes

- double **timestamp**

  Timestamp of the reading.
- std::vector< int > **sensor_ids**

  Camera ids for each of the images collected.
- std::vector< cv::Mat > **images**

  Raw image we have collected for each camera.
- std::vector< cv::Mat > **masks**

  Tracking masks for each camera we have.

13.6.1 Detailed Description

Struct for a collection of camera measurements.

For each image we have a camera id and timestamp that it occured at. If there are multiple cameras we will treat it as pair-wise stereo tracking.

13.7 ov_core::CamRadtan Class Reference

Radial-tangential / Brown–Conrady model pinhole camera model class.

```
#include <CamRadtan.h>
```

Public Member Functions

- CamRadtan (int width, int height)

  Default constructor.
- Eigen::Vector2f undistort_f (const Eigen::Vector2f &uv_dist) override

  Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.
- Eigen::Vector2f distort_f (const Eigen::Vector2f &uv_norm) override

  Given a normalized uv coordinate this will distort it to the raw image plane.
- void compute_distort_jacobian (const Eigen::Vector2d &uv_norm, Eigen::MatrixXd &H_dz_dzn, Eigen::MatrixXd &H_dz_dzeta) override

  Computes the derivative of raw distorted to normalized coordinate.

Additional Inherited Members

13.7.1 Detailed Description

Radial-tangential / Brown–Conrady model pinhole camera model class.

To calibrate camera intrinsics, we need to know how to map our normalized coordinates into the raw pixel coordinates on the image plane. We first employ the radial distortion as in `OpenCV model`:

$$\begin{bmatrix} u \\ v \end{bmatrix} := \mathbf{z}_k = \mathbf{h}_d(\mathbf{z}_{n,k},\ \boldsymbol{\zeta}) = \begin{bmatrix} f_x * x + c_x \\ f_y * y + c_y \end{bmatrix}$$

$$\begin{aligned} \text{where}\ \ x &= x_n(1 + k_1 r^2 + k_2 r^4) + 2p_1 x_n y_n + p_2(r^2 + 2x_n^2) \\ y &= y_n(1 + k_1 r^2 + k_2 r^4) + p_1(r^2 + 2y_n^2) + 2p_2 x_n y_n \\ r^2 &= x_n^2 + y_n^2 \end{aligned}$$

where $\mathbf{z}_{n,k} = [x_n \; y_n]^\top$ are the normalized coordinates of the 3D feature and u and v are the distorted image coordinates on the image plane. The following distortion and camera intrinsic (focal length and image center) parameters are involved in the above distortion model, which can be estimated online:

$$\boldsymbol{\zeta} = \begin{bmatrix} f_x & f_y & c_x & c_y & k_1 & k_2 & p_1 & p_2 \end{bmatrix}^\top$$

Note that we do not estimate the higher order (i.e., higher than fourth order) terms as in most offline calibration methods such as `Kalibr`. To estimate these intrinsic parameters (including the distortation parameters), the following Jacobian for these parameters is needed:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \boldsymbol{\zeta}} = \begin{bmatrix} x & 0 & 1 & 0 & f_x*(x_n r^2) & f_x*(x_n r^4) & f_x*(2x_n y_n) & f_x*(r^2+2x_n^2) \\ 0 & y & 0 & 1 & f_y*(y_n r^2) & f_y*(y_n r^4) & f_y*(r^2+2y_n^2) & f_y*(2x_n y_n) \end{bmatrix}$$

Similarly, the Jacobian with respect to the normalized coordinates can be obtained as follows:

$$\frac{\partial \mathbf{h}_d(\cdot)}{\partial \mathbf{z}_{n,k}} = \Big[\begin{matrix} f_x*((1+k_1r^2+k_2r^4)+(2k_1x_n^2+4k_2x_n^2(x_n^2+y_n^2))+2p_1y_n+(2p_2x_n+4p_2x_n)) & f_x*(2k_1x_ny_n+4 \\ f_y*(2k_1x_ny_n+4k_2x_ny_n(x_n^2+y_n^2)+2p_1x_n+2p_2y_n) & f_y*((1+k_1r^2+k_2r^4)+(2k_1y \end{matrix}$$

To equate this camera class to Kalibr's models, this is what you would use for `pinhole-radtan`.

13.7.2 Constructor & Destructor Documentation

13.7.2.1 CamRadtan()

```
ov_core::CamRadtan::CamRadtan (
              int width,
              int height )  [inline]
```

Default constructor.

Parameters

width	Width of the camera (raw pixels)
height	Height of the camera (raw pixels)

13.7.3 Member Function Documentation

13.7.3.1 compute_distort_jacobian()

```
void ov_core::CamRadtan::compute_distort_jacobian (
            const Eigen::Vector2d & uv_norm,
            Eigen::MatrixXd & H_dz_dzn,
            Eigen::MatrixXd & H_dz_dzeta )  [inline], [override], [virtual]
```

Computes the derivative of raw distorted to normalized coordinate.

Parameters

uv_norm	Normalized coordinates we wish to distort
H_dz_dzn	Derivative of measurement z in respect to normalized
H_dz_dzeta	Derivative of measurement z in respect to intrinsic parameters

Implements ov_core::CamBase.

13.7.3.2 distort_f()

```
Eigen::Vector2f ov_core::CamRadtan::distort_f (
            const Eigen::Vector2f & uv_norm )  [inline], [override], [virtual]
```

Given a normalized uv coordinate this will distort it to the raw image plane.

Parameters

uv_norm	Normalized coordinates we wish to distort

Returns

2d vector of raw uv coordinate

Implements ov_core::CamBase.

13.7.3.3 undistort_f()

```
Eigen::Vector2f ov_core::CamRadtan::undistort_f (
            const Eigen::Vector2f & uv_dist )  [inline], [override], [virtual]
```

Given a raw uv point, this will undistort it based on the camera matrices into normalized camera coords.

Parameters

uv_dist	Raw uv coordinate we wish to undistort

Returns

2d vector of normalized coordinates

Implements ov_core::CamBase.

13.8 ov_core::FeatureInitializer::ClonePose Struct Reference

Structure which stores pose estimates for use in triangulation.

```
#include <FeatureInitializer.h>
```

Public Member Functions

- **ClonePose** (const Eigen::Matrix< double, 3, 3 > &R, const Eigen::Matrix< double, 3, 1 > &p)
  Constructs pose from rotation and position.
- **ClonePose** (const Eigen::Matrix< double, 4, 1 > &q, const Eigen::Matrix< double, 3, 1 > &p)
  Constructs pose from quaternion and position.
- **ClonePose** ()
  Default constructor.
- const Eigen::Matrix< double, 3, 3 > & **Rot** ()
  Accessor for rotation.
- const Eigen::Matrix< double, 3, 1 > & **pos** ()
  Accessor for position.

Public Attributes

- Eigen::Matrix< double, 3, 3 > **_Rot**
  Rotation.
- Eigen::Matrix< double, 3, 1 > **_pos**
  Position.

13.8.1 Detailed Description

Structure which stores pose estimates for use in triangulation.

- R_GtoC - rotation from global to camera
- p_CinG - position of camera in global frame

13.9 ov_core::CpiBase Class Reference

Base class for continuous preintegration integrators.

```
#include <CpiBase.h>
```

Public Member Functions

- CpiBase (double sigma_w, double sigma_wb, double sigma_a, double sigma_ab, bool imu_avg_=false)

  Default constructor.
- void setLinearizationPoints (Eigen::Matrix< double, 3, 1 > b_w_lin_, Eigen::Matrix< double, 3, 1 > b_a_lin_, Eigen::Matrix< double, 4, 1 > q_k_lin_=Eigen::Matrix< double, 4, 1 >::Zero(), Eigen::Matrix< double, 3, 1 > grav_=Eigen::Matrix< double, 3, 1 >::Zero())

  Set linearization points of the integration.
- virtual void feed_IMU (double t_0, double t_1, Eigen::Matrix< double, 3, 1 > w_m_0, Eigen::Matrix< double, 3, 1 > a_m_0, Eigen::Matrix< double, 3, 1 > w_m_1=Eigen::Matrix< double, 3, 1 >::Zero(), Eigen::Matrix< double, 3, 1 > a_m_1=Eigen::Matrix< double, 3, 1 >::Zero())=0

  Main function that will sequentially compute the preintegration measurement.

Public Attributes

- bool **imu_avg** = false
- double **DT** = 0

  measurement integration time
- Eigen::Matrix< double, 3, 1 > **alpha_tau** = Eigen::Matrix<double, 3, 1>::Zero()

  alpha measurement mean
- Eigen::Matrix< double, 3, 1 > **beta_tau** = Eigen::Matrix<double, 3, 1>::Zero()

  beta measurement mean
- Eigen::Matrix< double, 4, 1 > **q_k2tau**

  orientation measurement mean
- Eigen::Matrix< double, 3, 3 > **R_k2tau** = Eigen::Matrix<double, 3, 3>::Identity()

  orientation measurement mean
- Eigen::Matrix< double, 3, 3 > **J_q** = Eigen::Matrix<double, 3, 3>::Zero()

  orientation Jacobian wrt b_w
- Eigen::Matrix< double, 3, 3 > **J_a** = Eigen::Matrix<double, 3, 3>::Zero()

  alpha Jacobian wrt b_w
- Eigen::Matrix< double, 3, 3 > **J_b** = Eigen::Matrix<double, 3, 3>::Zero()

  beta Jacobian wrt b_w
- Eigen::Matrix< double, 3, 3 > **H_a** = Eigen::Matrix<double, 3, 3>::Zero()

  alpha Jacobian wrt b_a
- Eigen::Matrix< double, 3, 3 > **H_b** = Eigen::Matrix<double, 3, 3>::Zero()

  beta Jacobian wrt b_a
- Eigen::Matrix< double, 3, 1 > **b_w_lin**

  b_w linearization point (gyroscope)
- Eigen::Matrix< double, 3, 1 > **b_a_lin**

  b_a linearization point (accelerometer)

- Eigen::Matrix< double, 4, 1 > **q_k_lin**

    q_k linearization point (only model 2 uses)
- Eigen::Matrix< double, 3, 1 > **grav** = Eigen::Matrix<double, 3, 1>::Zero()

    Global gravity.
- Eigen::Matrix< double, 12, 12 > **Q_c** = Eigen::Matrix<double, 12, 12>::Zero()

    Our continous-time measurement noise matrix (computed from contructor noise values)
- Eigen::Matrix< double, 15, 15 > **P_meas** = Eigen::Matrix<double, 15, 15>::Zero()

    Our final measurement covariance.
- Eigen::Matrix< double, 3, 3 > **eye3** = Eigen::Matrix<double, 3, 3>::Identity()
- Eigen::Matrix< double, 3, 1 > **e_1**
- Eigen::Matrix< double, 3, 1 > **e_2**
- Eigen::Matrix< double, 3, 1 > **e_3**
- Eigen::Matrix< double, 3, 3 > **e_1x**
- Eigen::Matrix< double, 3, 3 > **e_2x**
- Eigen::Matrix< double, 3, 3 > **e_3x**

13.9.1 Detailed Description

Base class for continuous preintegration integrators.

This is the base class that both continuous-time preintegrators extend. Please take a look at the derived classes CpiV1 and CpiV2 for the actual implementation. Please see the following publication for details on the theory Eckenhoff et al. [2019] :

> Continuous Preintegration Theory for Graph-based Visual-Inertial Navigation Authors: Kevin Eckenhoff, Patrick Geneva, and Guoquan Huang `http://udel.edu/~ghuang/papers/tr_cpi.pdf`

The steps to use this preintegration class are as follows:

1. call setLinearizationPoints() to set the bias/orientation linearization point
2. call feed_IMU() will all IMU measurements you want to precompound over
3. access public varibles, to get means, Jacobians, and measurement covariance

13.9.2 Constructor & Destructor Documentation

13.9.2.1 CpiBase()

```
ov_core::CpiBase::CpiBase (
             double sigma_w,
             double sigma_wb,
             double sigma_a,
             double sigma_ab,
             bool imu_avg_ = false )  [inline]
```

Default constructor.

Parameters

sigma_w	gyroscope white noise density (rad/s/sqrt(hz))
sigma_wb	gyroscope random walk (rad/s^2/sqrt(hz))
sigma_a	accelerometer white noise density (m/s^2/sqrt(hz))
sigma_ab	accelerometer random walk (m/s^3/sqrt(hz))
*imu_↩ avg_*	if we want to average the imu measurements (IJRR paper did not do this)

13.9.3 Member Function Documentation

13.9.3.1 feed_IMU()

```
virtual void ov_core::CpiBase::feed_IMU (
                double t_0,
                double t_1,
                Eigen::Matrix< double, 3, 1 > w_m_0,
                Eigen::Matrix< double, 3, 1 > a_m_0,
                Eigen::Matrix< double, 3, 1 > w_m_1 = Eigen::Matrix< double, 3, 1 >::Zero(),
                Eigen::Matrix< double, 3, 1 > a_m_1 = Eigen::Matrix< double, 3, 1 >::Zero() )  [pure
virtual]
```

Main function that will sequentially compute the preintegration measurement.

Parameters

in	*t_0*	first IMU timestamp
in	*t_1*	second IMU timestamp
in	*w_m↩ _0*	first imu gyroscope measurement
in	*a_m↩ _0*	first imu acceleration measurement
in	*w_m↩ _1*	second imu gyroscope measurement
in	*a_m↩ _1*	second imu acceleration measurement

This new IMU messages and will precompound our measurements, jacobians, and measurement covariance. Please see both CpiV1 and CpiV2 classes for implementation details on how this works.

Implemented in ov_core::CpiV1, and ov_core::CpiV2.

13.9.3.2 setLinearizationPoints()

```
void ov_core::CpiBase::setLinearizationPoints (
            Eigen::Matrix< double, 3, 1 > b_w_lin_,
            Eigen::Matrix< double, 3, 1 > b_a_lin_,
            Eigen::Matrix< double, 4, 1 > q_k_lin_ = Eigen::Matrix<double, 4, 1>::Zero(),
            Eigen::Matrix< double, 3, 1 > grav_ = Eigen::Matrix<double, 3, 1>::Zero() )  [inline]
```

Set linearization points of the integration.

Parameters

in	*b_w_lin_*	gyroscope bias linearization point
in	*b_a_lin_*	accelerometer bias linearization point
in	*q_k_lin_*	orientation linearization point (only model 2 uses)
in	*grav_*	global gravity at the current timestep

This function sets the linearization points we are to preintegrate about. For model 2 we will also pass the q_GtoK and current gravity estimate.

13.10 ov_core::CpiV1 Class Reference

Model 1 of continuous preintegration.

```
#include <CpiV1.h>
```

Public Member Functions

- CpiV1 (double sigma_w, double sigma_wb, double sigma_a, double sigma_ab, bool imu_avg_=false)

  Default constructor for our Model 1 preintegration (piecewise constant measurement assumption)
- void feed_IMU (double t_0, double t_1, Eigen::Matrix< double, 3, 1 > w_m_0, Eigen::Matrix< double, 3, 1 > a_m_0, Eigen::Matrix< double, 3, 1 > w_m_1=Eigen::Matrix< double, 3, 1 >::Zero(), Eigen::Matrix< double, 3, 1 > a_m_1=Eigen::Matrix< double, 3, 1 >::Zero())

  Our precompound function for Model 1.

Additional Inherited Members

13.10.1 Detailed Description

Model 1 of continuous preintegration.

This model is the "piecewise constant measurement assumption" which was first presented in:

> Eckenhoff, Kevin, Patrick Geneva, and Guoquan Huang. "High-accuracy preintegration for visual inertial navigation." International Workshop on the Algorithmic Foundations of Robotics. 2016.

Please see the following publication for details on the theory Eckenhoff et al. [2019] :

> Continuous Preintegration Theory for Graph-based Visual-Inertial Navigation Authors: Kevin Eckenhoff, Patrick Geneva, and Guoquan Huang `http://udel.edu/~ghuang/papers/tr_cpi.pdf`

The steps to use this preintegration class are as follows:

1. call setLinearizationPoints() to set the bias/orientation linearization point
2. call feed_IMU() will all IMU measurements you want to precompound over
3. access public varibles, to get means, Jacobians, and measurement covariance

13.10.2 Constructor & Destructor Documentation

13.10.2.1 CpiV1()

```
ov_core::CpiV1::CpiV1 (
            double sigma_w,
            double sigma_wb,
            double sigma_a,
            double sigma_ab,
            bool imu_avg_ = false )  [inline]
```

Default constructor for our Model 1 preintegration (piecewise constant measurement assumption)

Parameters

sigma_w	gyroscope white noise density (rad/s/sqrt(hz))
sigma_wb	gyroscope random walk (rad/s^2/sqrt(hz))
sigma_a	accelerometer white noise density (m/s^2/sqrt(hz))
sigma_ab	accelerometer random walk (m/s^3/sqrt(hz))
*imu_↩ avg_*	if we want to average the imu measurements (IJRR paper did not do this)

13.10.3 Member Function Documentation

13.10.3.1 feed_IMU()

```
void ov_core::CpiV1::feed_IMU (
              double t_0,
              double t_1,
              Eigen::Matrix< double, 3, 1 > w_m_0,
              Eigen::Matrix< double, 3, 1 > a_m_0,
              Eigen::Matrix< double, 3, 1 > w_m_1 = Eigen::Matrix<double, 3, 1>::Zero(),
              Eigen::Matrix< double, 3, 1 > a_m_1 = Eigen::Matrix<double, 3, 1>::Zero() )  [inline],
[virtual]
```

Our precompound function for Model 1.

Parameters

in	*t_0*	first IMU timestamp
in	*t_1*	second IMU timestamp
in	*w_m↩ _0*	first imu gyroscope measurement
in	*a_m↩ _0*	first imu acceleration measurement
in	*w_m↩ _1*	second imu gyroscope measurement
in	*a_m↩ _1*	second imu acceleration measurement

We will first analytically integrate our meansurements and Jacobians. Then we perform numerical integration for our measurement covariance.

Implements ov_core::CpiBase.

13.11 ov_core::CpiV2 Class Reference

Model 2 of continuous preintegration.

```
#include <CpiV2.h>
```

Public Member Functions

- CpiV2 (double sigma_w, double sigma_wb, double sigma_a, double sigma_ab, bool imu_avg_=false)

  Default constructor for our Model 2 preintegration (piecewise constant local acceleration assumption)
- void feed_IMU (double t_0, double t_1, Eigen::Matrix< double, 3, 1 > w_m_0, Eigen::Matrix< double, 3, 1 > a_m_0, Eigen::Matrix< double, 3, 1 > w_m_1=Eigen::Matrix< double, 3, 1 >::Zero(), Eigen::Matrix< double, 3, 1 > a_m_1=Eigen::Matrix< double, 3, 1 >::Zero())

  Our precompound function for Model 2.

Public Attributes

- bool **state_transition_jacobians** = true

  If we want to use analytical jacobians or not. In the paper we just numerically integrated the jacobians If set to false, we use a closed form version similar to model 1.
- Eigen::Matrix< double, 3, 3 > **O_a** = Eigen::Matrix<double, 3, 3>::Zero()
- Eigen::Matrix< double, 3, 3 > **O_b** = Eigen::Matrix<double, 3, 3>::Zero()

13.11.1 Detailed Description

Model 2 of continuous preintegration.

This model is the "piecewise constant local acceleration assumption." Please see the following publication for details on the theory Eckenhoff et al. [2019] :

> Continuous Preintegration Theory for Graph-based Visual-Inertial Navigation Authors: Kevin Eckenhoff, Patrick Geneva, and Guoquan Huang `http://udel.edu/~ghuang/papers/tr_cpi.pdf`

The steps to use this preintegration class are as follows:

1. call setLinearizationPoints() to set the bias/orientation linearization point
2. call feed_IMU() will all IMU measurements you want to precompound over
3. access public varibles, to get means, Jacobians, and measurement covariance

13.11.2 Constructor & Destructor Documentation

13.11.2.1 CpiV2()

```
ov_core::CpiV2::CpiV2 (
             double sigma_w,
             double sigma_wb,
             double sigma_a,
             double sigma_ab,
             bool imu_avg_ = false )  [inline]
```

Default constructor for our Model 2 preintegration (piecewise constant local acceleration assumption)

Parameters

sigma_w	gyroscope white noise density (rad/s/sqrt(hz))
sigma_wb	gyroscope random walk (rad/s^2/sqrt(hz))
sigma_a	accelerometer white noise density (m/s^2/sqrt(hz))
sigma_ab	accelerometer random walk (m/s^3/sqrt(hz))
*imu_↩ avg_*	if we want to average the imu measurements (IJRR paper did not do this)

13.11.3 Member Function Documentation

13.11.3.1 feed_IMU()

```
void ov_core::CpiV2::feed_IMU (
            double t_0,
            double t_1,
            Eigen::Matrix< double, 3, 1 > w_m_0,
            Eigen::Matrix< double, 3, 1 > a_m_0,
            Eigen::Matrix< double, 3, 1 > w_m_1 = Eigen::Matrix<double, 3, 1>::Zero(),
            Eigen::Matrix< double, 3, 1 > a_m_1 = Eigen::Matrix<double, 3, 1>::Zero() )  [inline],
[virtual]
```

Our precompound function for Model 2.

Parameters

in	*t_0*	first IMU timestamp
in	*t_1*	second IMU timestamp
in	*w_m↩_0*	first imu gyroscope measurement
in	*a_m↩_0*	first imu acceleration measurement
in	*w_m↩_1*	second imu gyroscope measurement
in	*a_m↩_1*	second imu acceleration measurement

We will first analytically integrate our meansurement. We can numerically or analytically integrate our bias jacobians. Then we perform numerical integration for our measurement covariance.

Implements ov_core::CpiBase.

13.12 ov_core::DatasetReader Class Reference

Helper functions to read in dataset files.

```
#include <dataset_reader.h>
```

Static Public Member Functions

- static void load_gt_file (std::string path, std::map< double, Eigen::Matrix< double, 17, 1 > > &gt_states)
  Load a ASL format groundtruth file.

- static bool get_gt_state (double timestep, Eigen::Matrix< double, 17, 1 > &imustate, std::map< double, Eigen↩ ::Matrix< double, 17, 1 > > &gt_states)

    Gets the 17x1 groundtruth state at a given timestep.
- static void load_simulated_trajectory (std::string path, std::vector< Eigen::VectorXd > &traj_data)

    This will load the trajectory into memory (space separated)

13.12.1 Detailed Description

Helper functions to read in dataset files.

This file has some nice functions for reading dataset files. One of the main datasets that we test against is the EuRoC MAV dataset. We have some nice utility functions here that handle loading of the groundtruth data. This can be used to initialize the system or for plotting and calculation of RMSE values without needing any alignment.

> M. Burri, J. Nikolic, P. Gohl, T. Schneider, J. Rehder, S. Omari,M. Achtelik and R. Siegwart, "↩ The EuRoC micro aerial vehicle datasets", International Journal of Robotic Research, DOI: 10.↩ 1177/0278364915620033, 2016. https://projects.asl.ethz.ch/datasets/doku.↩ php?id=kmavvisualinertialdatasets.

13.12.2 Member Function Documentation

13.12.2.1 get_gt_state()

```
static bool ov_core::DatasetReader::get_gt_state (
              double timestep,
              Eigen::Matrix< double, 17, 1 > & imustate,
              std::map< double, Eigen::Matrix< double, 17, 1 > > & gt_states )  [inline], [static]
```

Gets the 17x1 groundtruth state at a given timestep.

Parameters

timestep	timestep we want to get the groundtruth for
imustate	groundtruth state [time(sec),q_GtoI,p_IinG,v_IinG,b_gyro,b_accel]
gt_states	Should be loaded with groundtruth states, see load_gt_file() for details

Returns

true if we found the state, false otherwise

13.12.2.2 load_gt_file()

```
static void ov_core::DatasetReader::load_gt_file (
            std::string path,
            std::map< double, Eigen::Matrix< double, 17, 1 > > & gt_states ) [inline], [static]
```

Load a ASL format groundtruth file.

Parameters

path	Path to the CSV file of groundtruth data
gt_states	Will be filled with groundtruth states

Here we will try to load a groundtruth file that is in the ASL/EUROCMAV format. If we can't open the file, or it is in the wrong format we will error and exit the program. See get_gt_state() for a way to get the groundtruth state at a given timestep

13.12.2.3 load_simulated_trajectory()

```
static void ov_core::DatasetReader::load_simulated_trajectory (
            std::string path,
            std::vector< Eigen::VectorXd > & traj_data ) [inline], [static]
```

This will load the trajectory into memory (space separated)

Parameters

path	Path to the trajectory file that we want to read in.
traj_data	Will be filled with groundtruth states (timestamp(s), q_GtoI, p_IinG)

13.13 ov_core::Feature Class Reference

Sparse feature class used to collect measurements.

`#include <Feature.h>`

Public Member Functions

- void clean_old_measurements (const std::vector< double > &valid_times)

    Remove measurements that do not occur at passed timestamps.
- void clean_invalid_measurements (const std::vector< double > &invalid_times)

    Remove measurements that occur at the invalid timestamps.
- void clean_older_measurements (double timestamp)

    Remove measurements that are older then the specified timestamp.

Public Attributes

- size_t **featid**

  Unique ID of this feature.
- bool **to_delete**

  If this feature should be deleted.
- std::unordered_map< size_t, std::vector< Eigen::VectorXf > > **uvs**

  UV coordinates that this feature has been seen from (mapped by camera ID)
- std::unordered_map< size_t, std::vector< Eigen::VectorXf > > **uvs_norm**

  UV normalized coordinates that this feature has been seen from (mapped by camera ID)
- std::unordered_map< size_t, std::vector< double > > **timestamps**

  Timestamps of each UV measurement (mapped by camera ID)
- int **anchor_cam_id** = -1

  What camera ID our pose is anchored in!! By default the first measurement is the anchor.
- double **anchor_clone_timestamp**

  Timestamp of anchor clone.
- Eigen::Vector3d **p_FinA**

  Triangulated position of this feature, in the anchor frame.
- Eigen::Vector3d **p_FinG**

  Triangulated position of this feature, in the global frame.

13.13.1 Detailed Description

Sparse feature class used to collect measurements.

This feature class allows for holding of all tracking information for a given feature. Each feature has a unique ID assigned to it, and should have a set of feature tracks alongside it. See the FeatureDatabase class for details on how we load information into this, and how we delete features.

13.13.2 Member Function Documentation

13.13.2.1 clean_invalid_measurements()

```
void Feature::clean_invalid_measurements (
             const std::vector< double > & invalid_times )
```

Remove measurements that occur at the invalid timestamps.

Given a series of invalid timestamps, this will remove all measurements that have occurred at these times.

Parameters

invalid_times	Vector of timestamps that our measurements should not

13.13.2.2 clean_old_measurements()

```
void Feature::clean_old_measurements (
             const std::vector< double > & valid_times )
```

Remove measurements that do not occur at passed timestamps.

Given a series of valid timestamps, this will remove all measurements that have not occurred at these times. This would normally be used to ensure that the measurements that we have occur at our clone times.

Parameters

valid_times	Vector of timestamps that our measurements must occur at

13.13.2.3 clean_older_measurements()

```
void Feature::clean_older_measurements (
             double timestamp )
```

Remove measurements that are older then the specified timestamp.

Given a valid timestamp, this will remove all measurements that have occured earlier then this.

Parameters

timestamp	Timestamps that our measurements must occur after

13.14 ov_core::FeatureDatabase Class Reference

Database containing features we are currently tracking.

```
#include <FeatureDatabase.h>
```

Public Member Functions

- **FeatureDatabase** ()

  Default constructor.
- std::shared_ptr< Feature > get_feature (size_t id, bool remove=false)

Get a specified feature.
- void update_feature (size_t id, double timestamp, size_t cam_id, float u, float v, float u_n, float v_n)

  Update a feature object.
- std::vector< std::shared_ptr< Feature > > features_not_containing_newer (double timestamp, bool remove=false, bool skip_deleted=false)

  Get features that do not have newer measurement then the specified time.
- std::vector< std::shared_ptr< Feature > > features_containing_older (double timestamp, bool remove=false, bool skip_deleted=false)

  Get features that has measurements older then the specified time.
- std::vector< std::shared_ptr< Feature > > features_containing (double timestamp, bool remove=false, bool skip_deleted=false)

  Get features that has measurements at the specified time.
- void cleanup ()

  This function will delete all features that have been used up.
- void **cleanup_measurements** (double timestamp)

  This function will delete all feature measurements that are older then the specified timestamp.
- void **cleanup_measurements_exact** (double timestamp)

  This function will delete all feature measurements that are at the specified timestamp.
- size_t **size** ()

  Returns the size of the feature database.
- std::unordered_map< size_t, std::shared_ptr< Feature > > **get_internal_data** ()

  Returns the internal data (should not normally be used)
- void **append_new_measurements** (const std::shared_ptr< FeatureDatabase > &database)

  Will update the passed database with this database's latest feature information.

Protected Attributes

- std::mutex **mtx**

  Mutex lock for our map.
- std::unordered_map< size_t, std::shared_ptr< Feature > > **features_idlookup**

  Our lookup array that allow use to query based on ID.

13.14.1 Detailed Description

Database containing features we are currently tracking.

Each visual tracker has this database in it and it contains all features that we are tracking. The trackers will insert information into this database when they get new measurements from doing tracking. A user would then query this database for features that can be used for update and remove them after they have been processed.

A Note on Multi-Threading Support

There is some support for asynchronous multi-threaded access. Since each feature is a pointer just directly returning and using them is not thread safe. Thus, to be thread safe, use the "remove" flag for each function which will remove it from this feature database. This prevents the trackers from adding new measurements and editing the feature information. For example, if you are asynchronous tracking cameras and you chose to update the state, then remove all features you will use in update. The feature trackers will continue to add features while you update, whose measurements can be used in the next update step!

13.14.2 Member Function Documentation

13.14.2.1 cleanup()

```
void ov_core::FeatureDatabase::cleanup ( )  [inline]
```

This function will delete all features that have been used up.

If a feature was unable to be used, it will still remain since it will not have a delete flag set

13.14.2.2 features_containing()

```
std::vector< std::shared_ptr< Feature > > ov_core::FeatureDatabase::features_containing (
             double timestamp,
             bool remove = false,
             bool skip_deleted = false )  [inline]
```

Get features that has measurements at the specified time.

This function will return all features that have the specified time in them. This would be used to get all features that occurred at a specific clone/state.

13.14.2.3 features_containing_older()

```
std::vector< std::shared_ptr< Feature > > ov_core::FeatureDatabase::features_containing_older (
             double timestamp,
             bool remove = false,
             bool skip_deleted = false )  [inline]
```

Get features that has measurements older then the specified time.

This will collect all features that have measurements occurring before the specified timestamp. For example, we would want to remove all features older then the last clone/state in our sliding window.

13.14.2.4 features_not_containing_newer()

```
std::vector< std::shared_ptr< Feature > > ov_core::FeatureDatabase::features_not_containing_↩
newer (
             double timestamp,
             bool remove = false,
             bool skip_deleted = false )  [inline]
```

Get features that do not have newer measurement then the specified time.

This function will return all features that do not a measurement at a time greater than the specified time. For example this could be used to get features that have not been successfully tracked into the newest frame. All features returned will not have any measurements occurring at a time greater then the specified.

13.14.2.5 get_feature()

```
std::shared_ptr< Feature > ov_core::FeatureDatabase::get_feature (
             size_t id,
             bool remove = false )  [inline]
```

Get a specified feature.

Parameters

id	What feature we want to get
remove	Set to true if you want to remove the feature from the database (you will need to handle the freeing of memory)

Returns

Either a feature object, or null if it is not in the database.

13.14.2.6 update_feature()

```
void ov_core::FeatureDatabase::update_feature (
            size_t id,
            double timestamp,
            size_t cam_id,
            float u,
            float v,
            float u_n,
            float v_n )  [inline]
```

Update a feature object.

Parameters

id	ID of the feature we will update
timestamp	time that this measurement occured at
cam_id	which camera this measurement was from
u	raw u coordinate
v	raw v coordinate
u_n	undistorted/normalized u coordinate
v_n	undistorted/normalized v coordinate

This will update a given feature based on the passed ID it has. It will create a new feature, if it is an ID that we have not seen before.

13.15 ov_core::FeatureHelper Class Reference

Contains some nice helper functions for features.

```
#include <FeatureHelper.h>
```

Static Public Member Functions

- static void compute_disparity (std::shared_ptr< ov_core::FeatureDatabase > db, double time0, double time1, double &disp_mean, double &disp_var, int &total_feats)

  This functions will compute the disparity between common features in the two frames.

13.15.1 Detailed Description

Contains some nice helper functions for features.

These functions should only depend on feature and the feature database.

13.15.2 Member Function Documentation

13.15.2.1 compute_disparity()

```
static void ov_core::FeatureHelper::compute_disparity (
            std::shared_ptr< ov_core::FeatureDatabase > db,
            double time0,
            double time1,
            double & disp_mean,
            double & disp_var,
            int & total_feats )  [inline], [static]
```

This functions will compute the disparity between common features in the two frames.

First we find all features in the first frame. Then we loop through each and find the uv of it in the next requested frame. Features are skipped if no tracked feature is found (it was lost). NOTE: this is on the RAW coordinates of the feature not the normalized ones. NOTE: This computes the disparity over all cameras!

Parameters

db	Feature database pointer
time0	First camera frame timestamp
time1	Second camera frame timestamp
disp_mean	Average raw disparity
disp_var	Variance of the disparities
total_feats	Total number of common features

13.16 ov_core::FeatureInitializer Class Reference

Class that triangulates feature.

```
#include <FeatureInitializer.h>
```

Classes

- struct ClonePose

  Structure which stores pose estimates for use in triangulation.

Public Member Functions

- FeatureInitializer (FeatureInitializerOptions &options)

  Default constructor.
- bool single_triangulation (Feature ∗feat, std::unordered_map< size_t, std::unordered_map< double, ClonePose > > &clonesCAM)

  Uses a linear triangulation to get initial estimate for the feature.
- bool single_triangulation_1d (Feature ∗feat, std::unordered_map< size_t, std::unordered_map< double, ClonePose > > &clonesCAM)

  Uses a linear triangulation to get initial estimate for the feature, treating the anchor observation as a true bearing.
- bool single_gaussnewton (Feature ∗feat, std::unordered_map< size_t, std::unordered_map< double, ClonePose > > &clonesCAM)

  Uses a nonlinear triangulation to refine initial linear estimate of the feature.
- const FeatureInitializerOptions config ()

  Gets the current configuration of the feature initializer.

Protected Member Functions

- double compute_error (std::unordered_map< size_t, std::unordered_map< double, ClonePose > > &clones↩ CAM, Feature ∗feat, double alpha, double beta, double rho)

  Helper function for the gauss newton method that computes error of the given estimate.

Protected Attributes

- FeatureInitializerOptions **_options**

  Contains options for the initializer process.

13.16.1 Detailed Description

Class that triangulates feature.

This class has the functions needed to triangulate and then refine a given 3D feature. As in the standard MSCKF, we know the clones of the camera from propagation and past updates. Thus, we just need to triangulate a feature in 3D with the known poses and then refine it. One should first call the single_triangulation() function afterwhich single_gaussnewton() allows for refinement. Please see the Feature Triangulation page for detailed derivations.

13.16.2 Constructor & Destructor Documentation

13.16.2.1 FeatureInitializer()

```
ov_core::FeatureInitializer::FeatureInitializer (
            FeatureInitializerOptions & options )  [inline]
```

Default constructor.

Parameters

options	Options for the initializer

13.16.3 Member Function Documentation

13.16.3.1 compute_error()

```
double FeatureInitializer::compute_error (
            std::unordered_map< size_t, std::unordered_map< double, ClonePose > > & clonesCAM,
            Feature * feat,
            double alpha,
            double beta,
            double rho )  [protected]
```

Helper function for the gauss newton method that computes error of the given estimate.

Parameters

clonesCAM	Map between camera ID to map of timestamp to camera pose estimate
feat	Pointer to the feature
alpha	x/z in anchor
beta	y/z in anchor
rho	1/z inverse depth

13.16.3.2 config()

```
const FeatureInitializerOptions ov_core::FeatureInitializer::config ( )  [inline]
```

Gets the current configuration of the feature initializer.

Returns

Const feature initializer config

13.16.3.3 single_gaussnewton()

```
bool FeatureInitializer::single_gaussnewton (
            Feature * feat,
            std::unordered_map< size_t, std::unordered_map< double, ClonePose > > & clonesCAM )
```

Uses a nonlinear triangulation to refine initial linear estimate of the feature.

Parameters

feat	Pointer to feature
clonesCAM	Map between camera ID to map of timestamp to camera pose estimate (rotation from global to camera, position of camera in global frame)

Returns

Returns false if it fails to be optimize (based on the thresholds)

13.16.3.4 single_triangulation()

```
bool FeatureInitializer::single_triangulation (
            Feature * feat,
            std::unordered_map< size_t, std::unordered_map< double, ClonePose > > & clonesCAM )
```

Uses a linear triangulation to get initial estimate for the feature.

The derivations for this method can be found in the 3D Cartesian Triangulation documentation page.

Parameters

feat	Pointer to feature
clonesCAM	Map between camera ID to map of timestamp to camera pose estimate (rotation from global to camera, position of camera in global frame)

Returns

Returns false if it fails to triangulate (based on the thresholds)

13.16.3.5 single_triangulation_1d()

```
bool FeatureInitializer::single_triangulation_1d (
            Feature * feat,
            std::unordered_map< size_t, std::unordered_map< double, ClonePose > > & clonesCAM )
```

Uses a linear triangulation to get initial estimate for the feature, treating the anchor observation as a true bearing.

The derivations for this method can be found in the 1D Depth Triangulation documentation page. This function should be used if you want speed, or know your anchor bearing is reasonably accurate.

Parameters

feat	Pointer to feature
clonesCAM	Map between camera ID to map of timestamp to camera pose estimate (rotation from global to camera, position of camera in global frame)

Returns

Returns false if it fails to triangulate (based on the thresholds)

13.17 ov_core::FeatureInitializerOptions Struct Reference

Struct which stores all our feature initializer options.

```
#include <FeatureInitializerOptions.h>
```

Public Member Functions

- void **print** (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  Nice print function of what parameters we have loaded.

Public Attributes

- bool **triangulate_1d** = false

  If we should perform 1d triangulation instead of 3d.
- bool **refine_features** = true

  If we should perform Levenberg-Marquardt refinment.
- int **max_runs** = 5

  Max runs for Levenberg-Marquardt.
- double **init_lamda** = 1e-3

  Init lambda for Levenberg-Marquardt optimization.
- double **max_lamda** = 1e10

  Max lambda for Levenberg-Marquardt optimization.
- double **min_dx** = 1e-6

Cutoff for dx increment to consider as converged.

- double **min_dcost** = 1e-6

  Cutoff for cost decrement to consider as converged.
- double **lam_mult** = 10

  Multiplier to increase/decrease lambda.
- double **min_dist** = 0.10

  Minimum distance to accept triangulated features.
- double **max_dist** = 60

  Minimum distance to accept triangulated features.
- double **max_baseline** = 40

  Max baseline ratio to accept triangulated features.
- double **max_cond_number** = 10000

  Max condition number of linear triangulation matrix accept triangulated features.

13.17.1 Detailed Description

Struct which stores all our feature initializer options.

13.18 ov_core::Grider_FAST Class Reference

Extracts FAST features in a grid pattern.

```
#include <Grider_FAST.h>
```

Static Public Member Functions

- static bool compare_response (cv::KeyPoint first, cv::KeyPoint second)

  Compare keypoints based on their response value.
- static void perform_griding (const cv::Mat &img, const cv::Mat &mask, std::vector< cv::KeyPoint > &pts, int num↩ _features, int grid_x, int grid_y, int threshold, bool nonmaxSuppression)

  This function will perform grid extraction using FAST.

13.18.1 Detailed Description

Extracts FAST features in a grid pattern.

As compared to just extracting fast features over the entire image, we want to have as uniform of extractions as possible over the image plane. Thus we split the image into a bunch of small grids, and extract points in each. We then pick enough top points in each grid so that we have the total number of desired points.

13.18.2 Member Function Documentation

13.18.2.1 compare_response()

```
static bool ov_core::Grider_FAST::compare_response (
             cv::KeyPoint first,
             cv::KeyPoint second )  [inline], [static]
```

Compare keypoints based on their response value.

Parameters

first	First keypoint
second	Second keypoint

We want to have the keypoints with the highest values! See: https://stackoverflow.com/a/10910921

13.18.2.2 perform_griding()

```
static void ov_core::Grider_FAST::perform_griding (
             const cv::Mat & img,
             const cv::Mat & mask,
             std::vector< cv::KeyPoint > & pts,
             int num_features,
             int grid_x,
             int grid_y,
             int threshold,
             bool nonmaxSuppression )  [inline], [static]
```

This function will perform grid extraction using FAST.

Parameters

img	Image we will do FAST extraction on
mask	Region of the image we do not want to extract features in (255 = do not detect features)
pts	vector of extracted points we will return
num_features	max number of features we want to extract
grid_x	size of grid in the x-direction / u-direction
grid_y	size of grid in the y-direction / v-direction
threshold	FAST threshold paramter (10 is a good value normally)
nonmaxSuppression	if FAST should perform non-max suppression (true normally)

Given a specified grid size, this will try to extract fast features from each grid. It will then return the best from each grid in the return vector.

13.19 ov_type::IMU Class Reference

Derived Type class that implements an IMU state.

```
#include <IMU.h>
```

Public Member Functions

- void set_local_id (int new_id) override

  Sets id used to track location of variable in the filter covariance.
- void update (const Eigen::VectorXd &dx) override

  Performs update operation using JPLQuat update for orientation, then vector updates for position, velocity, gyro bias, and accel bias (in that order).
- void set_value (const Eigen::MatrixXd &new_value) override

  Sets the value of the estimate.
- void set_fej (const Eigen::MatrixXd &new_value) override

  Sets the value of the first estimate.
- std::shared_ptr< Type > clone () override

  Create a clone of this variable.
- std::shared_ptr< Type > check_if_subvariable (const std::shared_ptr< Type > check) override

  Determine if pass variable is a sub-variable.
- Eigen::Matrix< double, 3, 3 > **Rot** () const

  Rotation access.
- Eigen::Matrix< double, 3, 3 > **Rot_fej** () const

  FEJ Rotation access.
- Eigen::Matrix< double, 4, 1 > **quat** () const

  Rotation access quaternion.
- Eigen::Matrix< double, 4, 1 > **quat_fej** () const

  FEJ Rotation access quaternion.
- Eigen::Matrix< double, 3, 1 > **pos** () const

  Position access.
- Eigen::Matrix< double, 3, 1 > **pos_fej** () const

  FEJ position access.
- Eigen::Matrix< double, 3, 1 > **vel** () const

  Velocity access.
- Eigen::Matrix< double, 3, 1 > **vel_fej** () const
- Eigen::Matrix< double, 3, 1 > **bias_g** () const

  Gyro bias access.
- Eigen::Matrix< double, 3, 1 > **bias_g_fej** () const

  FEJ gyro bias access.
- Eigen::Matrix< double, 3, 1 > **bias_a** () const

  Accel bias access.
- Eigen::Matrix< double, 3, 1 > **bias_a_fej** () const
- std::shared_ptr< PoseJPL > **pose** ()

  Pose type access.
- std::shared_ptr< JPLQuat > **q** ()

  Quaternion type access.
- std::shared_ptr< Vec > **p** ()

  Position type access.
- std::shared_ptr< Vec > **v** ()

  Velocity type access.
- std::shared_ptr< Vec > **bg** ()

  Gyroscope bias access.
- std::shared_ptr< Vec > **ba** ()

  Acceleration bias access.

Protected Member Functions

- void set_value_internal (const Eigen::MatrixXd &new_value)

  Sets the value of the estimate.
- void set_fej_internal (const Eigen::MatrixXd &new_value)

  Sets the value of the first estimate.

Protected Attributes

- std::shared_ptr< PoseJPL > **_pose**

  Pose subvariable.
- std::shared_ptr< Vec > **_v**

  Velocity subvariable.
- std::shared_ptr< Vec > **_bg**

  Gyroscope bias subvariable.
- std::shared_ptr< Vec > **_ba**

  Acceleration bias subvariable.

13.19.1 Detailed Description

Derived Type class that implements an IMU state.

Contains a PoseJPL, Vec velocity, Vec gyro bias, and Vec accel bias. This should be similar to that of the standard MSCKF state besides the ordering. The pose is first, followed by velocity, etc.

13.19.2 Member Function Documentation

13.19.2.1 check_if_subvariable()

```
std::shared_ptr< Type > ov_type::IMU::check_if_subvariable (
             const std::shared_ptr< Type > check )  [inline], [override], [virtual]
```

Determine if pass variable is a sub-variable.

If the passed variable is a sub-variable or the current variable this will return it. Otherwise it will return a nullptr, meaning that it was unable to be found.

Parameters

check	Type pointer to compare our subvariables to

Reimplemented from ov_type::Type.

13.19.2.2 clone()

```
std::shared_ptr< Type > ov_type::IMU::clone ( )  [inline], [override], [virtual]
```

Create a clone of this variable.

Implements ov_type::Type.

13.19.2.3 set_fej()

```
void ov_type::IMU::set_fej (
             const Eigen::MatrixXd & new_value )  [inline], [override], [virtual]
```

Sets the value of the first estimate.

Parameters

new_value	New value we should set

Reimplemented from ov_type::Type.

13.19.2.4 set_fej_internal()

```
void ov_type::IMU::set_fej_internal (
             const Eigen::MatrixXd & new_value )  [inline], [protected]
```

Sets the value of the first estimate.

Parameters

new_value	New value we should set

13.19.2.5 set_local_id()

```
void ov_type::IMU::set_local_id (
             int new_id )  [inline], [override], [virtual]
```

Sets id used to track location of variable in the filter covariance.

Note that we update the sub-variables also.

Parameters

new←_id	entry in filter covariance corresponding to this variable

Reimplemented from ov_type::Type.

13.19.2.6 set_value()

```
void ov_type::IMU::set_value (
             const Eigen::MatrixXd & new_value )  [inline], [override], [virtual]
```

Sets the value of the estimate.

Parameters

new_value	New value we should set

Reimplemented from ov_type::Type.

13.19.2.7 set_value_internal()

```
void ov_type::IMU::set_value_internal (
             const Eigen::MatrixXd & new_value )  [inline], [protected]
```

Sets the value of the estimate.

Parameters

new_value	New value we should set

13.19.2.8 update()

```
void ov_type::IMU::update (
             const Eigen::VectorXd & dx )  [inline], [override], [virtual]
```

Performs update operation using JPLQuat update for orientation, then vector updates for position, velocity, gyro bias, and accel bias (in that order).

Parameters

dx	15 DOF vector encoding update using the following order (q, p, v, bg, ba)

Implements ov_type::Type.

13.20 ov_core::ImuData Struct Reference

Struct for a single imu measurement (time, wm, am)

```
#include <sensor_data.h>
```

Public Member Functions

- bool **operator**< (const ImuData &other) const

  Sort function to allow for using of STL containers.

Public Attributes

- double **timestamp**

  Timestamp of the reading.
- Eigen::Matrix< double, 3, 1 > **wm**

  Gyroscope reading, angular velocity (rad/s)
- Eigen::Matrix< double, 3, 1 > **am**

  Accelerometer reading, linear acceleration (m/s^2)

13.20.1 Detailed Description

Struct for a single imu measurement (time, wm, am)

13.21 ov_init::InertialInitializer Class Reference

Initializer for visual-inertial system.

```
#include <InertialInitializer.h>
```

Public Member Functions

- InertialInitializer (InertialInitializerOptions &params_, std::shared_ptr< ov_core::FeatureDatabase > db)

  Default constructor.
- void feed_imu (const ov_core::ImuData &message)

  Feed function for inertial data.
- bool initialize (double &timestamp, Eigen::MatrixXd &covariance, std::vector< std::shared_ptr< ov_type::Type > > &order, std::shared_ptr< ov_type::IMU > t_imu, bool wait_for_jerk=true)

  Try to get the initialized system.

Protected Attributes

- InertialInitializerOptions **params**

  Initialization parameters.
- std::shared_ptr< ov_core::FeatureDatabase > **_db**

  Feature tracker database with all features in it.
- std::shared_ptr< std::vector< ov_core::ImuData > > **imu_data**

  Our history of IMU messages (time, angular, linear)
- std::shared_ptr< StaticInitializer > **init_static**

  Static initialization helper class.

13.21.1 Detailed Description

Initializer for visual-inertial system.

This will try to do both dynamic and state initialization of the state. The user can request to wait for a jump in our IMU readings (i.e. device is picked up) or to initialize as soon as possible. For state initialization, the user needs to specify the calibration beforehand, otherwise dynamic is always used.

The logic is as follows:

1. Try to perform dynamic initialization of state elements.
2. If this fails and we have calibration then we can try to do static initialization
3. If the unit is stationary and we are waiting for a jerk, just return, otherwise initialize the state!

The dynamic system is based on an implementation and extension of the work `Estimator initialization in vision-aided inertial navigation with unknown camera-IMU calibration` Dong-Si and Mourikis [2012] which solves the initialization problem by first creating a linear system for recovering the camera to IMU rotation, then for velocity, gravity, and feature positions, and finally a full optimization to allow for covariance recovery.

13.21.2 Constructor & Destructor Documentation

13.21.2.1 InertialInitializer()

```
InertialInitializer::InertialInitializer (
            InertialInitializerOptions & params_,
            std::shared_ptr< ov_core::FeatureDatabase > db )  [explicit]
```

Default constructor.

Parameters

params←↩ _	Parameters loaded from either ROS or CMDLINE
db	Feature tracker database with all features in it

13.21.3 Member Function Documentation

13.21.3.1 feed_imu()

```
void ov_init::InertialInitializer::feed_imu (
              const ov_core::ImuData & message )  [inline]
```

Feed function for inertial data.

Parameters

message	Contains our timestamp and inertial information

13.21.3.2 initialize()

```
bool InertialInitializer::initialize (
              double & timestamp,
              Eigen::MatrixXd & covariance,
              std::vector< std::shared_ptr< ov_type::Type > > & order,
              std::shared_ptr< ov_type::IMU > t_imu,
              bool wait_for_jerk = true )
```

Try to get the initialized system.

Parameters

out	*timestamp*	Timestamp we have initialized the state at
out	*covariance*	Calculated covariance of the returned state
out	*order*	Order of the covariance matrix
out	*t_imu*	Our imu type (need to have correct ids)
	wait_for_jerk	If true we will wait for a "jerk"

Returns

True if we have successfully initialized our system

13.22 ov_init::InertialInitializerOptions Struct Reference

Struct which stores all options needed for state estimation.

```
#include <InertialInitializerOptions.h>
```

Public Member Functions

- void print_and_load (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)
  This function will load the non-simulation parameters of the system and print.
- void print_and_load_initializer (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)
  This function will load print out all initializer settings loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.
- void print_and_load_state (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)
  This function will load and print all state parameters (e.g. sensor extrinsics) This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Public Attributes

- double **init_window_time** = 1.0
  Amount of time we will initialize over (seconds)
- double **init_imu_thresh** = 1.0
  Variance threshold on our acceleration to be classified as moving.
- double **init_max_disparity** = 1.0
  Max disparity we will consider the unit to be stationary.
- int **init_max_features** = 20
  Number of features we should try to track.
- double **gravity_mag** = 9.81
  Gravity magnitude in the global frame (i.e. should be 9.81 typically)
- int **num_cameras** = 1
  Number of distinct cameras that we will observe features in.

13.22.1 Detailed Description

Struct which stores all options needed for state estimation.

This is broken into a few different parts: estimator, trackers, and simulation. If you are going to add a parameter here you will need to add it to the parsers. You will also need to add it to the print statement at the bottom of each.

13.22.2 Member Function Documentation

13.22.2.1 print_and_load()

```
void ov_init::InertialInitializerOptions::print_and_load (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load the non-simulation parameters of the system and print.

Parameters

parser	If not null, this parser will be used to load our parameters

13.22.2.2 print_and_load_initializer()

```
void ov_init::InertialInitializerOptions::print_and_load_initializer (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load print out all initializer settings loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.22.2.3 print_and_load_state()

```
void ov_init::InertialInitializerOptions::print_and_load_state (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load and print all state parameters (e.g. sensor extrinsics) This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.23 ov_type::JPLQuat Class Reference

Derived Type class that implements JPL quaternion.

```
#include <JPLQuat.h>
```

Public Member Functions

- void update (const Eigen::VectorXd &dx) override

  Implements update operation by left-multiplying the current quaternion with a quaternion built from a small axis-angle perturbation.
- void set_value (const Eigen::MatrixXd &new_value) override

  Sets the value of the estimate and recomputes the internal rotation matrix.
- void set_fej (const Eigen::MatrixXd &new_value) override

  Sets the fej value and recomputes the fej rotation matrix.
- std::shared_ptr< Type > clone () override

  Create a clone of this variable.
- Eigen::Matrix< double, 3, 3 > **Rot** () const

  Rotation access.
- Eigen::Matrix< double, 3, 3 > **Rot_fej** () const

  FEJ Rotation access.

Protected Member Functions

- void set_value_internal (const Eigen::MatrixXd &new_value)

  Sets the value of the estimate and recomputes the internal rotation matrix.
- void set_fej_internal (const Eigen::MatrixXd &new_value)

  Sets the fej value and recomputes the fej rotation matrix.

Protected Attributes

- Eigen::Matrix< double, 3, 3 > **_R**
- Eigen::Matrix< double, 3, 3 > **_Rfej**

13.23.1 Detailed Description

Derived Type class that implements JPL quaternion.

This quaternion uses a left-multiplicative error state and follows the "JPL convention". Please checkout our utility functions in the quat_ops.h file. We recommend that people new quaternions check out the following resources:

- `http://mars.cs.umn.edu/tr/reports/Trawny05b.pdf`
- `ftp://naif.jpl.nasa.gov/pub/naif/misc/Quaternion_White_Paper/Quaternions←_White_Paper.pdf`

13.23.2 Member Function Documentation

13.23.2.1 clone()

```
std::shared_ptr< Type > ov_type::JPLQuat::clone ( ) [inline], [override], [virtual]
```

Create a clone of this variable.

Implements ov_type::Type.

13.23.2.2 set_fej()

```
void ov_type::JPLQuat::set_fej (
            const Eigen::MatrixXd & new_value ) [inline], [override], [virtual]
```

Sets the fej value and recomputes the fej rotation matrix.

Parameters

new_value	New value for the quaternion estimate

Reimplemented from ov_type::Type.

13.23.2.3 set_fej_internal()

```
void ov_type::JPLQuat::set_fej_internal (
            const Eigen::MatrixXd & new_value ) [inline], [protected]
```

Sets the fej value and recomputes the fej rotation matrix.

Parameters

new_value	New value for the quaternion estimate

13.23.2.4 set_value()

```
void ov_type::JPLQuat::set_value (
            const Eigen::MatrixXd & new_value )  [inline], [override], [virtual]
```

Sets the value of the estimate and recomputes the internal rotation matrix.

Parameters

new_value	New value for the quaternion estimate

Reimplemented from ov_type::Type.

13.23.2.5 set_value_internal()

```
void ov_type::JPLQuat::set_value_internal (
            const Eigen::MatrixXd & new_value )  [inline], [protected]
```

Sets the value of the estimate and recomputes the internal rotation matrix.

Parameters

new_value	New value for the quaternion estimate

13.23.2.6 update()

```
void ov_type::JPLQuat::update (
            const Eigen::VectorXd & dx )  [inline], [override], [virtual]
```

Implements update operation by left-multiplying the current quaternion with a quaternion built from a small axis-angle perturbation.

$$\bar{q} = norm\left(\begin{bmatrix} 0.5 * \theta_{\mathbf{dx}} \\ 1 \end{bmatrix}\right)\hat{\bar{q}}$$

Parameters

dx	Axis-angle representation of the perturbing quaternion

Implements ov_type::Type.

13.24 ov_core::LambdaBody Class Reference

Helper class to do OpenCV parallelization.

```
#include <opencv_lambda_body.h>
```

Public Member Functions

- **LambdaBody** (const std::function< void(const cv::Range &)> &body)
- void **operator()** (const cv::Range &range) const override

13.24.1 Detailed Description

Helper class to do OpenCV parallelization.

This is a utility class required to build with older version of opencv On newer versions this doesn't seem to be needed, but here we just use it to ensure we can work for more opencv version. https://answers.↩opencv.org/question/65800/how-to-use-lambda-as-a-parameter-to-parallel_for_↩/?answer=130691#post-id-130691

13.25 ov_type::Landmark Class Reference

Type that implements a persistent SLAM feature.

```
#include <Landmark.h>
```

Public Member Functions

- **Landmark** (int dim)

  Default constructor (feature is a Vec of size 3 or Vec of size 1)
- void update (const Eigen::VectorXd &dx) override

  Overrides the default vector update rule We want to selectively update the FEJ value if we are using an anchored representation.
- Eigen::Matrix< double, 3, 1 > get_xyz (bool getfej) const

  Will return the position of the feature in the global frame of reference.
- void set_from_xyz (Eigen::Matrix< double, 3, 1 > p_FinG, bool isfej)

  Will set the current value based on the representation.

Public Attributes

- size_t **_featid**

    Feature ID of this landmark (corresponds to frontend id)
- int **_unique_camera_id** = -1

    What unique camera stream this slam feature was observed from.
- int **_anchor_cam_id** = -1

    What camera ID our pose is anchored in!! By default the first measurement is the anchor.
- double **_anchor_clone_timestamp** = -1

    Timestamp of anchor clone.
- bool **has_had_anchor_change** = false

    Boolean if this landmark has had at least one anchor change.
- bool **should_marg** = false

    Boolean if this landmark should be marginalized out.
- Eigen::Vector3d **uv_norm_zero**

    First normalized uv coordinate bearing of this measurement (used for single depth representation)
- Eigen::Vector3d **uv_norm_zero_fej**

    First estimate normalized uv coordinate bearing of this measurement (used for single depth representation)
- LandmarkRepresentation::Representation **_feat_representation**

    What feature representation this feature currently has.

Additional Inherited Members

13.25.1 Detailed Description

Type that implements a persistent SLAM feature.

We store the feature ID that should match the IDs in the trackers. Additionally if this is an anchored representation we store what clone timestamp this is anchored from and what camera. If this features should be marginalized its flag can be set and during cleanup it will be removed.

13.25.2 Member Function Documentation

13.25.2.1 get_xyz()

```
Eigen::Matrix< double, 3, 1 > Landmark::get_xyz (
             bool getfej ) const
```

Will return the position of the feature in the global frame of reference.

Parameters

getfej	Set to true to get the landmark FEJ value

Returns

Position of feature either in global or anchor frame

13.25.2.2 set_from_xyz()

```
void Landmark::set_from_xyz (
             Eigen::Matrix< double, 3, 1 > p_FinG,
             bool isfej )
```

Will set the current value based on the representation.

Parameters

p_FinG	Position of the feature either in global or anchor frame
isfej	Set to true to set the landmark FEJ value

13.25.2.3 update()

```
void ov_type::Landmark::update (
             const Eigen::VectorXd & dx )  [inline], [override], [virtual]
```

Overrides the default vector update rule We want to selectively update the FEJ value if we are using an anchored representation.

Parameters

dx	Additive error state correction

Implements ov_type::Type.

13.26 ov_type::LandmarkRepresentation Class Reference

Class has useful feature representation types.

```
#include <LandmarkRepresentation.h>
```

Public Types

- enum Representation {
  GLOBAL_3D , **GLOBAL_FULL_INVERSE_DEPTH** , **ANCHORED_3D** , **ANCHORED_FULL_INVERSE_↩ DEPTH** ,
  ANCHORED_MSCKF_INVERSE_DEPTH , **ANCHORED_INVERSE_DEPTH_SINGLE** , **UNKNOWN** }

  What feature representation our state can use.

Static Public Member Functions

- static std::string as_string (Representation feat_representation)

  Returns a string representation of this enum value. Used to debug print out what the user has selected as the representation.
- static Representation from_string (const std::string &feat_representation)

  Returns a string representation of this enum value. Used to debug print out what the user has selected as the representation.
- static bool is_relative_representation (Representation feat_representation)

  Helper function that checks if the passed feature representation is a relative or global.

13.26.1 Detailed Description

Class has useful feature representation types.

13.26.2 Member Function Documentation

13.26.2.1 as_string()

```
static std::string ov_type::LandmarkRepresentation::as_string (
            Representation feat_representation )  [inline], [static]
```

Returns a string representation of this enum value. Used to debug print out what the user has selected as the representation.

Parameters

feat_representation	Representation we want to check

Returns

String version of the passed enum

13.26.2.2 from_string()

```
static Representation ov_type::LandmarkRepresentation::from_string (
            const std::string & feat_representation )  [inline], [static]
```

Returns a string representation of this enum value. Used to debug print out what the user has selected as the representation.

Parameters

feat_representation	String we want to find the enum of

Returns

Representation, will be "unknown" if we coun't parse it

13.26.2.3 is_relative_representation()

```
static bool ov_type::LandmarkRepresentation::is_relative_representation (
            Representation feat_representation )  [inline], [static]
```

Helper function that checks if the passed feature representation is a relative or global.

Parameters

feat_representation	Representation we want to check

Returns

True if it is a relative representation

13.27 ov_eval::Loader Class Reference

Has helper functions to load text files from disk and process them.

```
#include <Loader.h>
```

Static Public Member Functions

- static void load_data (std::string path_traj, std::vector< double > &times, std::vector< Eigen::Matrix< double, 7, 1 > > &poses, std::vector< Eigen::Matrix3d > &cov_ori, std::vector< Eigen::Matrix3d > &cov_pos)

  This will load space separated trajectory into memory.

- static void load_data_csv (std::string path_traj, std::vector< double > &times, std::vector< Eigen::Matrix< double, 7, 1 > > &poses, std::vector< Eigen::Matrix3d > &cov_ori, std::vector< Eigen::Matrix3d > &cov_pos)

    This will load comma separated trajectory into memory (ASL/ETH format)
- static void load_simulation (std::string path, std::vector< Eigen::VectorXd > &values)

    Load an arbitrary sized row of space separated values, used for our simulation.
- static void load_timing_flamegraph (std::string path, std::vector< std::string > &names, std::vector< double > &times, std::vector< Eigen::VectorXd > &timing_values)

    Load comma separated timing file from pid_ros.py file.
- static void load_timing_percent (std::string path, std::vector< double > &times, std::vector< Eigen::Vector3d > &summed_values, std::vector< Eigen::VectorXd > &node_values)

    Load space separated timing file from pid_ros.py file.
- static double get_total_length (const std::vector< Eigen::Matrix< double, 7, 1 > > &poses)

    Will calculate the total trajectory distance.

13.27.1 Detailed Description

Has helper functions to load text files from disk and process them.

13.27.2 Member Function Documentation

13.27.2.1 get_total_length()

```
double Loader::get_total_length (
             const std::vector< Eigen::Matrix< double, 7, 1 > > & poses )  [static]
```

Will calculate the total trajectory distance.

Parameters

poses	Pose at every timestep [pos,quat]

Returns

Distance travels (meters)

13.27.2.2 load_data()

```
void Loader::load_data (
             std::string path_traj,
             std::vector< double > & times,
             std::vector< Eigen::Matrix< double, 7, 1 > > & poses,
             std::vector< Eigen::Matrix3d > & cov_ori,
             std::vector< Eigen::Matrix3d > & cov_pos )  [static]
```

This will load *space* separated trajectory into memory.

Parameters

path_traj	Path to the trajectory file that we want to read in.
times	Timesteps in seconds for each pose
poses	Pose at every timestep [pos,quat]
cov_ori	Vector of orientation covariances at each timestep (empty if we can't load)
cov_pos	Vector of position covariances at each timestep (empty if we can't load)

13.27.2.3 load_data_csv()

```
void Loader::load_data_csv (
            std::string path_traj,
            std::vector< double > & times,
            std::vector< Eigen::Matrix< double, 7, 1 > > & poses,
            std::vector< Eigen::Matrix3d > & cov_ori,
            std::vector< Eigen::Matrix3d > & cov_pos )  [static]
```

This will load *comma* separated trajectory into memory (ASL/ETH format)

Parameters

path_traj	Path to the trajectory file that we want to read in.
times	Timesteps in seconds for each pose
poses	Pose at every timestep [pos,quat]
cov_ori	Vector of orientation covariances at each timestep (empty if we can't load)
cov_pos	Vector of position covariances at each timestep (empty if we can't load)

13.27.2.4 load_simulation()

```
void Loader::load_simulation (
            std::string path,
            std::vector< Eigen::VectorXd > & values )  [static]
```

Load an arbitrary sized row of *space* separated values, used for our simulation.

Parameters

path	Path to our text file to load
values	Each row of values

13.27.2.5 load_timing_flamegraph()

```
void Loader::load_timing_flamegraph (
             std::string path,
             std::vector< std::string > & names,
             std::vector< double > & times,
             std::vector< Eigen::VectorXd > & timing_values )  [static]
```

Load *comma* separated timing file from pid_ros.py file.

Parameters

path	Path to our text file to load
names	Names of each timing category
times	Timesteps in seconds for each measurement
timing_values	Component timing values for the given timestamp

13.27.2.6 load_timing_percent()

```
void Loader::load_timing_percent (
             std::string path,
             std::vector< double > & times,
             std::vector< Eigen::Vector3d > & summed_values,
             std::vector< Eigen::VectorXd > & node_values )  [static]
```

Load space separated timing file from pid_ros.py file.

Parameters

path	Path to our text file to load
times	Timesteps in seconds for each measurement
summed_values	Summed node values [cpu,mem,num_threads]
node_values	Values for each separate node [cpu,mem,num_threads]

13.28 ov_msckf::Propagator::NoiseManager Struct Reference

Struct of our imu noise parameters.

```
#include <Propagator.h>
```

Public Member Functions

- void **print** ()

  Nice print function of what parameters we have loaded.

Public Attributes

- double **sigma_w** = 1.6968e-04

  Gyroscope white noise (rad/s/sqrt(hz))
- double **sigma_w_2** = pow(1.6968e-04, 2)

  Gyroscope white noise covariance.
- double **sigma_wb** = 1.9393e-05

  Gyroscope random walk (rad/s^2/sqrt(hz))
- double **sigma_wb_2** = pow(1.9393e-05, 2)

  Gyroscope random walk covariance.
- double **sigma_a** = 2.0000e-3

  Accelerometer white noise (m/s^2/sqrt(hz))
- double **sigma_a_2** = pow(2.0000e-3, 2)

  Accelerometer white noise covariance.
- double **sigma_ab** = 3.0000e-03

  Accelerometer random walk (m/s^3/sqrt(hz))
- double **sigma_ab_2** = pow(3.0000e-03, 2)

  Accelerometer random walk covariance.

13.28.1 Detailed Description

Struct of our imu noise parameters.

13.29 ov_type::PoseJPL Class Reference

Derived Type class that implements a 6 d.o.f pose.

```
#include <PoseJPL.h>
```

Public Member Functions

- void set_local_id (int new_id) override

  Sets id used to track location of variable in the filter covariance.
- void update (const Eigen::VectorXd &dx) override

  Update q and p using a the JPLQuat update for orientation and vector update for position.
- void set_value (const Eigen::MatrixXd &new_value) override

  Sets the value of the estimate.
- void set_fej (const Eigen::MatrixXd &new_value) override

  Sets the value of the first estimate.
- std::shared_ptr< Type > clone () override

  Create a clone of this variable.
- std::shared_ptr< Type > check_if_subvariable (const std::shared_ptr< Type > check) override

  Determine if pass variable is a sub-variable.
- Eigen::Matrix< double, 3, 3 > **Rot** () const

  Rotation access.
- Eigen::Matrix< double, 3, 3 > **Rot_fej** () const

  FEJ Rotation access.
- Eigen::Matrix< double, 4, 1 > **quat** () const

  Rotation access as quaternion.
- Eigen::Matrix< double, 4, 1 > **quat_fej** () const

  FEJ Rotation access as quaternion.
- Eigen::Matrix< double, 3, 1 > **pos** () const

  Position access.
- Eigen::Matrix< double, 3, 1 > **pos_fej** () const
- std::shared_ptr< JPLQuat > **q** ()
- std::shared_ptr< Vec > **p** ()

Protected Member Functions

- void set_value_internal (const Eigen::MatrixXd &new_value)

  Sets the value of the estimate.
- void set_fej_internal (const Eigen::MatrixXd &new_value)

  Sets the value of the first estimate.

Protected Attributes

- std::shared_ptr< JPLQuat > **_q**

  Subvariable containing orientation.
- std::shared_ptr< Vec > **_p**

  Subvariable containing position.

13.29.1 Detailed Description

Derived Type class that implements a 6 d.o.f pose.

Internally we use a JPLQuat quaternion representation for the orientation and 3D Vec position. Please see JPLQuat for details on its update procedure and its left multiplicative error.

13.29.2 Member Function Documentation

13.29.2.1 check_if_subvariable()

```
std::shared_ptr< Type > ov_type::PoseJPL::check_if_subvariable (
            const std::shared_ptr< Type > check )  [inline], [override], [virtual]
```

Determine if pass variable is a sub-variable.

If the passed variable is a sub-variable or the current variable this will return it. Otherwise it will return a nullptr, meaning that it was unable to be found.

Parameters

check	Type pointer to compare our subvariables to

Reimplemented from ov_type::Type.

13.29.2.2 clone()

```
std::shared_ptr< Type > ov_type::PoseJPL::clone ( )  [inline], [override], [virtual]
```

Create a clone of this variable.

Implements ov_type::Type.

13.29.2.3 set_fej()

```
void ov_type::PoseJPL::set_fej (
            const Eigen::MatrixXd & new_value )  [inline], [override], [virtual]
```

Sets the value of the first estimate.

Parameters

new_value	New value we should set

Reimplemented from ov_type::Type.

13.29.2.4 set_fej_internal()

```
void ov_type::PoseJPL::set_fej_internal (
            const Eigen::MatrixXd & new_value )  [inline], [protected]
```

Sets the value of the first estimate.

Parameters

new_value	New value we should set

13.29.2.5 set_local_id()

```
void ov_type::PoseJPL::set_local_id (
            int new_id )  [inline], [override], [virtual]
```

Sets id used to track location of variable in the filter covariance.

Note that we update the sub-variables also.

Parameters

new←_id	entry in filter covariance corresponding to this variable

Reimplemented from ov_type::Type.

13.29.2.6 set_value()

```
void ov_type::PoseJPL::set_value (
            const Eigen::MatrixXd & new_value )  [inline], [override], [virtual]
```

Sets the value of the estimate.

Parameters

new_value	New value we should set

Reimplemented from ov_type::Type.

13.29.2.7 set_value_internal()

```
void ov_type::PoseJPL::set_value_internal (
             const Eigen::MatrixXd & new_value )  [inline], [protected]
```

Sets the value of the estimate.

Parameters

new_value	New value we should set

13.29.2.8 update()

```
void ov_type::PoseJPL::update (
             const Eigen::VectorXd & dx )  [inline], [override], [virtual]
```

Update q and p using a the JPLQuat update for orientation and vector update for position.

Parameters

dx	Correction vector (orientation then position)

Implements ov_type::Type.

13.30 ov_core::Printer Class Reference

Printer for open_vins that allows for various levels of printing to be done.

`#include <print.h>`

Public Types

- enum PrintLevel {
  ALL = 0 , **DEBUG** = 1 , **INFO** = 2 , **WARNING** = 3 ,
  ERROR = 4 , **SILENT** = 5 }

  The different print levels possible.

Static Public Member Functions

- static void setPrintLevel (const std::string &level)

  Set the print level to use for all future printing to stdout.
- static void setPrintLevel (PrintLevel level)

  Set the print level to use for all future printing to stdout.
- static void debugPrint (PrintLevel level, const char location[ ], const char line[ ], const char ∗format,...)

  The print function that prints to stdout.

Static Public Attributes

- static PrintLevel **current_print_level** = PrintLevel::INFO

  The current print level.

13.30.1 Detailed Description

Printer for open_vins that allows for various levels of printing to be done.

To set the global verbosity level one can do the following:

```
ov_core::Printer::setPrintLevel("WARNING");
ov_core::Printer::setPrintLevel(ov_core::Printer::PrintLevel::WARNING);
```

13.30.2 Member Enumeration Documentation

13.30.2.1 PrintLevel

```
enum ov_core::Printer::PrintLevel
```

The different print levels possible.

- PrintLevel::ALL : All PRINT_XXXX will output to the console
- PrintLevel::DEBUG : "DEBUG", "INFO", "WARNING" and "ERROR" will be printed. "ALL" will be silenced
- PrintLevel::INFO : "INFO", "WARNING" and "ERROR" will be printed. "ALL" and "DEBUG" will be silenced
- PrintLevel::WARNING : "WARNING" and "ERROR" will be printed. "ALL", "DEBUG" and "INFO" will be silenced
- PrintLevel::ERROR : Only "ERROR" will be printed. All the rest are silenced
- PrintLevel::SILENT : All PRINT_XXXX will be silenced.

13.30.3 Member Function Documentation

13.30.3.1 debugPrint()

```
void Printer::debugPrint (
            PrintLevel level,
            const char location[],
            const char line[],
            const char * format,
             ... ) [static]
```

The print function that prints to stdout.

Parameters

level	the print level for this print call
location	the location the print was made from
line	the line the print was made from
format	The printf format

13.30.3.2 setPrintLevel() [1/2]

```
void Printer::setPrintLevel (
            const std::string & level )  [static]
```

Set the print level to use for all future printing to stdout.

Parameters

level	The debug level to use

13.30.3.3 setPrintLevel() [2/2]

```
void Printer::setPrintLevel (
            PrintLevel level )  [static]
```

Set the print level to use for all future printing to stdout.

Parameters

level	The debug level to use

13.31 ov_msckf::Propagator Class Reference

Performs the state covariance and mean propagation using imu measurements.

```
#include <Propagator.h>
```

Classes

- struct NoiseManager

    Struct of our imu noise parameters.

Public Member Functions

- Propagator (NoiseManager noises, double gravity_mag)

  Default constructor.
- void feed_imu (const ov_core::ImuData &message)

  Stores incoming inertial readings.
- void propagate_and_clone (std::shared_ptr< State > state, double timestamp)

  Propagate state up to given timestamp and then clone.
- void fast_state_propagate (std::shared_ptr< State > state, double timestamp, Eigen::Matrix< double, 13, 1 > &state_plus)

  Gets what the state and its covariance will be at a given timestamp.

Static Public Member Functions

- static std::vector< ov_core::ImuData > select_imu_readings (const std::vector< ov_core::ImuData > &imu_data, double time0, double time1, bool warn=true)

  Helper function that given current imu data, will select imu readings between the two times.
- static ov_core::ImuData interpolate_data (const ov_core::ImuData &imu_1, const ov_core::ImuData &imu_2, double timestamp)

  Nice helper function that will linearly interpolate between two imu messages.

Protected Member Functions

- void predict_and_compute (std::shared_ptr< State > state, const ov_core::ImuData &data_minus, const ov_core::ImuData &data_plus, Eigen::Matrix< double, 15, 15 > &F, Eigen::Matrix< double, 15, 15 > &Qd)

  Propagates the state forward using the imu data and computes the noise covariance and state-transition matrix of this interval.
- void predict_mean_discrete (std::shared_ptr< State > state, double dt, const Eigen::Vector3d &w_hat1, const Eigen::Vector3d &a_hat1, const Eigen::Vector3d &w_hat2, const Eigen::Vector3d &a_hat2, Eigen::Vector4d &new_q, Eigen::Vector3d &new_v, Eigen::Vector3d &new_p)

  Discrete imu mean propagation.
- void predict_mean_rk4 (std::shared_ptr< State > state, double dt, const Eigen::Vector3d &w_hat1, const Eigen↩ ::Vector3d &a_hat1, const Eigen::Vector3d &w_hat2, const Eigen::Vector3d &a_hat2, Eigen::Vector4d &new_q, Eigen::Vector3d &new_v, Eigen::Vector3d &new_p)

  RK4 imu mean propagation.

Protected Attributes

- double **last_prop_time_offset** = 0.0

  Estimate for time offset at last propagation time.
- bool **have_last_prop_time_offset** = false
- NoiseManager **_noises**

  Container for the noise values.
- std::vector< ov_core::ImuData > **imu_data**

  Our history of IMU messages (time, angular, linear)
- Eigen::Vector3d **_gravity**

  Gravity vector.

13.31.1 Detailed Description

Performs the state covariance and mean propagation using imu measurements.

We will first select what measurements we need to propagate with. We then compute the state transition matrix at each step and update the state and covariance. For derivations look at IMU Propagation Derivations page which has detailed equations.

13.31.2 Constructor & Destructor Documentation

13.31.2.1 Propagator()

```
ov_msckf::Propagator::Propagator (
            NoiseManager noises,
            double gravity_mag )  [inline]
```

Default constructor.

Parameters

noises	imu noise characteristics (continuous time)
gravity_mag	Global gravity magnitude of the system (normally 9.81)

13.31.3 Member Function Documentation

13.31.3.1 fast_state_propagate()

```
void Propagator::fast_state_propagate (
            std::shared_ptr< State > state,
            double timestamp,
            Eigen::Matrix< double, 13, 1 > & state_plus )
```

Gets what the state and its covariance will be at a given timestamp.

This can be used to find what the state will be in the "future" without propagating it. This will propagate a clone of the current IMU state and its covariance matrix. This is typically used to provide high frequency pose estimates between updates.

Parameters

state	Pointer to state
timestamp	Time to propagate to
state_plus	The propagated state (q_GtoI, p_IinG, v_IinG, w_IinI)

13.31.3.2 feed_imu()

```
void ov_msckf::Propagator::feed_imu (
            const ov_core::ImuData & message )  [inline]
```

Stores incoming inertial readings.

Parameters

message	Contains our timestamp and inertial information

13.31.3.3 interpolate_data()

```
static ov_core::ImuData ov_msckf::Propagator::interpolate_data (
            const ov_core::ImuData & imu_1,
            const ov_core::ImuData & imu_2,
            double timestamp )  [inline], [static]
```

Nice helper function that will linearly interpolate between two imu messages.

This should be used instead of just "cutting" imu messages that bound the camera times Give better time offset if we use this function, could try other orders/splines if the imu is slow.

Parameters

imu_1	imu at begining of interpolation interval
imu_2	imu at end of interpolation interval
timestamp	Timestamp being interpolated to

13.31.3.4 predict_and_compute()

```
void Propagator::predict_and_compute (
            std::shared_ptr< State > state,
            const ov_core::ImuData & data_minus,
            const ov_core::ImuData & data_plus,
            Eigen::Matrix< double, 15, 15 > & F,
            Eigen::Matrix< double, 15, 15 > & Qd )  [protected]
```

Propagates the state forward using the imu data and computes the noise covariance and state-transition matrix of this interval.

This function can be replaced with analytical/numerical integration or when using a different state representation. This contains our state transition matrix along with how our noise evolves in time. If you have other state variables besides the IMU that evolve you would add them here. See the Discrete-time Error-state Propagation page for details on how this was derived.

Parameters

state	Pointer to state
data_minus	imu readings at beginning of interval
data_plus	imu readings at end of interval
F	State-transition matrix over the interval
Qd	Discrete-time noise covariance over the interval

13.31.3.5 predict_mean_discrete()

```
void Propagator::predict_mean_discrete (
                std::shared_ptr< State > state,
                double dt,
                const Eigen::Vector3d & w_hat1,
                const Eigen::Vector3d & a_hat1,
                const Eigen::Vector3d & w_hat2,
                const Eigen::Vector3d & a_hat2,
                Eigen::Vector4d & new_q,
                Eigen::Vector3d & new_v,
                Eigen::Vector3d & new_p )  [protected]
```

Discrete imu mean propagation.

See IMU Propagation Derivations for details on these equations.

$$
\begin{aligned}
{}^{I_{k+1}}_{G}\hat{\bar{q}} &= \exp\left(\frac{1}{2}\boldsymbol{\Omega}\big(\boldsymbol{\omega}_{m,k}-\hat{\mathbf{b}}_{g,k}\big)\Delta t\right){}^{I_k}_{G}\hat{\bar{q}} \\
{}^{G}\hat{\mathbf{v}}_{k+1} &= {}^{G}\hat{\mathbf{v}}_{I_k} - {}^{G}\mathbf{g}\Delta t + {}^{I_k}_{G}\hat{\mathbf{R}}^\top(\mathbf{a}_{m,k}-\hat{\mathbf{b}}_{\mathbf{a},k})\Delta t \\
{}^{G}\hat{\mathbf{p}}_{I_{k+1}} &= {}^{G}\hat{\mathbf{p}}_{I_k} + {}^{G}\hat{\mathbf{v}}_{I_k}\Delta t - \frac{1}{2}{}^{G}\mathbf{g}\Delta t^2 + \frac{1}{2}{}^{I_k}_{G}\hat{\mathbf{R}}^\top(\mathbf{a}_{m,k}-\hat{\mathbf{b}}_{\mathbf{a},k})\Delta t^2
\end{aligned}
$$

Parameters

state	Pointer to state
dt	Time we should integrate over

Parameters

w_hat1	Angular velocity with bias removed
a_hat1	Linear acceleration with bias removed
w_hat2	Next angular velocity with bias removed
a_hat2	Next linear acceleration with bias removed
new←_q	The resulting new orientation after integration
new←_v	The resulting new velocity after integration
new←_p	The resulting new position after integration

13.31.3.6 predict_mean_rk4()

```
void Propagator::predict_mean_rk4 (
            std::shared_ptr< State > state,
            double dt,
            const Eigen::Vector3d & w_hat1,
            const Eigen::Vector3d & a_hat1,
            const Eigen::Vector3d & w_hat2,
            const Eigen::Vector3d & a_hat2,
            Eigen::Vector4d & new_q,
            Eigen::Vector3d & new_v,
            Eigen::Vector3d & new_p )  [protected]
```

RK4 imu mean propagation.

See this wikipedia page on `Runge-Kutta Methods`. We are doing a RK4 method, `this wolframe page` has the forth order equation defined below. We define function $f(t,y)$ where y is a function of time t, see IMU Kinematics for the definition of the continous-time functions.

$$
\begin{aligned}
k_1 &= f(t_0, y_0)\Delta t \\
k_2 &= f(t_0 + \frac{\Delta t}{2}, y_0 + \frac{1}{2}k_1)\Delta t \\
k_3 &= f(t_0 + \frac{\Delta t}{2}, y_0 + \frac{1}{2}k_2)\Delta t \\
k_4 &= f(t_0 + \Delta t, y_0 + k_3)\Delta t \\
y_{0+\Delta t} &= y_0 + \left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right)
\end{aligned}
$$

Parameters

state	Pointer to state
dt	Time we should integrate over

Parameters

w_hat1	Angular velocity with bias removed
a_hat1	Linear acceleration with bias removed
w_hat2	Next angular velocity with bias removed
a_hat2	Next linear acceleration with bias removed
new←_q	The resulting new orientation after integration
new←_v	The resulting new velocity after integration
new←_p	The resulting new position after integration

13.31.3.7 propagate_and_clone()

```
void Propagator::propagate_and_clone (
            std::shared_ptr< State > state,
            double timestamp )
```

Propagate state up to given timestamp and then clone.

This will first collect all imu readings that occured between the *current* state time and the new time we want the state to be at. If we don't have any imu readings we will try to extrapolate into the future. After propagating the mean and covariance using our dynamics, We clone the current imu pose as a new clone in our state.

Parameters

state	Pointer to state
timestamp	Time to propagate to and clone at

13.31.3.8 select_imu_readings()

```
std::vector< ov_core::ImuData > Propagator::select_imu_readings (
            const std::vector< ov_core::ImuData > & imu_data,
            double time0,
            double time1,
            bool warn = true ) [static]
```

Helper function that given current imu data, will select imu readings between the two times.

This will create measurements that we will integrate with, and an extra measurement at the end. We use the interpolate_data() function to "cut" the imu readings at the begining and end of the integration. The timestamps passed should already take into account the time offset values.

Parameters

imu_data	IMU data we will select measurements from
time0	Start timestamp
time1	End timestamp
warn	If we should warn if we don't have enough IMU to propagate with (e.g. fast prop will get warnings otherwise)

Returns

Vector of measurements (if we could compute them)

13.32 ov_eval::Recorder Class Reference

This class takes in published poses and writes them to file.

```
#include <Recorder.h>
```

Public Member Functions

- Recorder (std::string filename)

  Default constructor that will open the specified file on disk. If the output directory does not exists this will also create the directory path to this file.
- void callback_odometry (const nav_msgs::OdometryPtr &msg)

  Callback for nav_msgs::Odometry message types.
- void callback_pose (const geometry_msgs::PoseStampedPtr &msg)

  Callback for geometry_msgs::PoseStamped message types.
- void callback_posecovariance (const geometry_msgs::PoseWithCovarianceStampedPtr &msg)

  Callback for geometry_msgs::PoseWithCovarianceStamped message types.
- void callback_transform (const geometry_msgs::TransformStampedPtr &msg)

  Callback for geometry_msgs::TransformStamped message types.

Protected Member Functions

- void **write** ()

  This is the main write function that will save to disk. This should be called after we have saved the desired pose to our class variables.

Protected Attributes

- std::ofstream **outfile**
- bool **has_covariance** = false
- double **timestamp**
- Eigen::Vector4d **q_ItoG**
- Eigen::Vector3d **p_IinG**
- Eigen::Matrix< double, 3, 3 > **cov_rot**
- Eigen::Matrix< double, 3, 3 > **cov_pos**

13.32.1 Detailed Description

This class takes in published poses and writes them to file.

Original code is based on this modified posemsg_to_file. Output is in a text file that is space deliminated and can be read by all scripts. If we have a covariance then we also save the upper triangular part to file so we can calculate NEES values.

13.32.2 Constructor & Destructor Documentation

13.32.2.1 Recorder()

```
ov_eval::Recorder::Recorder (
             std::string filename )  [inline]
```

Default constructor that will open the specified file on disk. If the output directory does not exists this will also create the directory path to this file.

Parameters

filename	Desired file we want to "record" into

13.32.3 Member Function Documentation

13.32.3.1 callback_odometry()

```
void ov_eval::Recorder::callback_odometry (
             const nav_msgs::OdometryPtr & msg )  [inline]
```

Callback for nav_msgs::Odometry message types.

Note that covariance is in the order (x, y, z, rotation about X axis, rotation about Y axis, rotation about Z axis). http://docs.ros.org/api/geometry_msgs/html/msg/PoseWithCovariance.html

Parameters

msg	New message

13.32.3.2 callback_pose()

```
void ov_eval::Recorder::callback_pose (
             const geometry_msgs::PoseStampedPtr & msg )  [inline]
```

Callback for geometry_msgs::PoseStamped message types.

Parameters

msg	New message

13.32.3.3 callback_posecovariance()

```
void ov_eval::Recorder::callback_posecovariance (
             const geometry_msgs::PoseWithCovarianceStampedPtr & msg )  [inline]
```

Callback for geometry_msgs::PoseWithCovarianceStamped message types.

Note that covariance is in the order (x, y, z, rotation about X axis, rotation about Y axis, rotation about Z axis). http://docs.ros.org/api/geometry_msgs/html/msg/PoseWithCovariance.html

Parameters

msg	New message

13.32.3.4 callback_transform()

```
void ov_eval::Recorder::callback_transform (
             const geometry_msgs::TransformStampedPtr & msg )  [inline]
```

Callback for geometry_msgs::TransformStamped message types.

Parameters

msg	New message

13.33 ov_eval::ResultSimulation Class Reference

A single simulation run (the full state not just pose).

```
#include <ResultSimulation.h>
```

Public Member Functions

- ResultSimulation (std::string path_est, std::string path_std, std::string path_gt)
  Default constructor that will load our data from file.
- void plot_state (bool doplotting, double max_time=INFINITY)
  Will plot the state error and its three sigma bounds.
- void plot_timeoff (bool doplotting, double max_time=INFINITY)
  Will plot the state imu camera offset and its sigma bound.
- void plot_cam_instrinsics (bool doplotting, double max_time=INFINITY)
  Will plot the camera calibration intrinsics.
- void plot_cam_extrinsics (bool doplotting, double max_time=INFINITY)
  Will plot the camera calibration extrinsic transform.

Protected Attributes

- std::vector< Eigen::VectorXd > **est_state**
- std::vector< Eigen::VectorXd > **gt_state**
- std::vector< Eigen::VectorXd > **state_cov**

13.33.1 Detailed Description

A single simulation run (the full state not just pose).

This should match the recording logic that is in the ov_msckf::RosVisualizer in which we write both estimate, their deviation, and groundtruth to three files. We enforce that these files first contain the current IMU state, then time offset, number of cameras, then the camera calibration states. If we are not performing calibration these should all be written to file, just their deviation should be zero as they are 100% certain.

13.33.2 Constructor & Destructor Documentation

13.33.2.1 ResultSimulation()

```
ResultSimulation::ResultSimulation (
            std::string path_est,
            std::string path_std,
            std::string path_gt )
```

Default constructor that will load our data from file.

Parameters

path_est	Path to the estimate text file
path_std	Path to the standard deviation file
path_gt	Path to the groundtruth text file

Assert they are of equal length

13.33.3 Member Function Documentation

13.33.3.1 plot_cam_extrinsics()

```
void ResultSimulation::plot_cam_extrinsics (
             bool doplotting,
             double max_time = INFINITY )
```

Will plot the camera calibration extrinsic transform.

Parameters

doplotting	True if you want to display the plots
max_time	Max number of second we want to plot

13.33.3.2 plot_cam_instrinsics()

```
void ResultSimulation::plot_cam_instrinsics (
             bool doplotting,
             double max_time = INFINITY )
```

Will plot the camera calibration intrinsics.

Parameters

doplotting	True if you want to display the plots
max_time	Max number of second we want to plot

13.33.3.3 plot_state()

```
void ResultSimulation::plot_state (
            bool doplotting,
            double max_time = INFINITY )
```

Will plot the state error and its three sigma bounds.

Parameters

doplotting	True if you want to display the plots
max_time	Max number of second we want to plot

13.33.3.4 plot_timeoff()

```
void ResultSimulation::plot_timeoff (
            bool doplotting,
            double max_time = INFINITY )
```

Will plot the state imu camera offset and its sigma bound.

Parameters

doplotting	True if you want to display the plots
max_time	Max number of second we want to plot

13.34 ov_eval::ResultTrajectory Class Reference

A single run for a given dataset.

```
#include <ResultTrajectory.h>
```

Public Member Functions

- ResultTrajectory (std::string path_est, std::string path_gt, std::string alignment_method)

  Default constructor that will load, intersect, and align our trajectories.
- void calculate_ate (Statistics &error_ori, Statistics &error_pos)

  Computes the Absolute Trajectory Error (ATE) for this trajectory.
- void calculate_ate_2d (Statistics &error_ori, Statistics &error_pos)

  Computes the Absolute Trajectory Error (ATE) for this trajectory in the 2d x-y plane.

- void calculate_rpe (const std::vector< double > &segment_lengths, std::map< double, std::pair< Statistics, Statistics > > &error_rpe)

  Computes the Relative Pose Error (RPE) for this trajectory.
- void calculate_nees (Statistics &nees_ori, Statistics &nees_pos)

  Computes the Normalized Estimation Error Squared (NEES) for this trajectory.
- void calculate_error (Statistics &posx, Statistics &posy, Statistics &posz, Statistics &orix, Statistics &oriy, Statistics &oriz, Statistics &roll, Statistics &pitch, Statistics &yaw)

  Computes the error at each timestamp for this trajectory.

Protected Member Functions

- std::vector< int > compute_comparison_indices_length (std::vector< double > &distances, double distance, double max_dist_diff)

  Gets the indices at the end of subtractories of a given length when starting at each index. For each starting pose, find the end pose index which is the desired distance away.

Protected Attributes

- std::vector< double > **est_times**
- std::vector< double > **gt_times**
- std::vector< Eigen::Matrix< double, 7, 1 > > **est_poses**
- std::vector< Eigen::Matrix< double, 7, 1 > > **gt_poses**
- std::vector< Eigen::Matrix3d > **est_covori**
- std::vector< Eigen::Matrix3d > **est_covpos**
- std::vector< Eigen::Matrix3d > **gt_covori**
- std::vector< Eigen::Matrix3d > **gt_covpos**
- std::vector< Eigen::Matrix< double, 7, 1 > > **est_poses_aignedtoGT**
- std::vector< Eigen::Matrix< double, 7, 1 > > **gt_poses_aignedtoEST**

13.34.1 Detailed Description

A single run for a given dataset.

This class has all the error function which can be calculated for the loaded trajectory. Given a groundtruth and trajectory we first align the two so that they are in the same frame. From there the following errors could be computed:

- Absolute trajectory error
- Relative pose Error
- Normalized estimation error squared
- Error and bound at each timestep

Please see the System Evaluation page for details and Zhang and Scaramuzza `A Tutorial on Quantitative Trajectory Evaluation for Visual(-Inertial) Odometry` paper for implementation specific details.

13.34.2 Constructor & Destructor Documentation

13.34.2.1 ResultTrajectory()

```
ResultTrajectory::ResultTrajectory (
            std::string path_est,
            std::string path_gt,
            std::string alignment_method )
```

Default constructor that will load, intersect, and align our trajectories.

Parameters

path_est	Path to the estimate text file
path_gt	Path to the groundtruth text file
alignment_method	The alignment method to use [sim3, se3, posyaw, none]

13.34.3 Member Function Documentation

13.34.3.1 calculate_ate()

```
void ResultTrajectory::calculate_ate (
            Statistics & error_ori,
            Statistics & error_pos )
```

Computes the Absolute Trajectory Error (ATE) for this trajectory.

This will first do our alignment of the two trajectories. Then at each point the error will be calculated and normed as follows:

$$e_{ATE} = \sqrt{\frac{1}{K}\sum_{k=1}^{K} ||\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}^{+}_{k,i}||_2^2}$$

Parameters

error_ori	Error values for the orientation
error_pos	Error values for the position

13.34.3.2 calculate_ate_2d()

```
void ResultTrajectory::calculate_ate_2d (
            Statistics & error_ori,
            Statistics & error_pos )
```

Computes the Absolute Trajectory Error (ATE) for this trajectory in the 2d x-y plane.

This will first do our alignment of the two trajectories. We just grab the yaw component of the orientation and the xy plane error. Then at each point the error will be calculated and normed as follows:

$$e_{ATE} = \sqrt{\frac{1}{K}\sum_{k=1}^{K}||\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}^{+}_{k,i}||_2^2}$$

Parameters

error_ori	Error values for the orientation (yaw error)
error_pos	Error values for the position (xy error)

13.34.3.3 calculate_error()

```
void ResultTrajectory::calculate_error (
            Statistics & posx,
            Statistics & posy,
            Statistics & posz,
            Statistics & orix,
            Statistics & oriy,
            Statistics & oriz,
            Statistics & roll,
            Statistics & pitch,
            Statistics & yaw )
```

Computes the error at each timestamp for this trajectory.

As compared to ATE error (see calculate_ate()) this will compute the error for each individual pose component. This is normally used if you just want to look at a single run on a single dataset.

$$e_{rmse,k} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}||\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i}||_2^2}$$

Parameters

posx	Position x-axis error and bound if we have it in our file
posy	Position y-axis error and bound if we have it in our file

Parameters

posz	Position z-axis error and bound if we have it in our file
orix	Orientation x-axis error and bound if we have it in our file
oriy	Orientation y-axis error and bound if we have it in our file
oriz	Orientation z-axis error and bound if we have it in our file
roll	Orientation roll error and bound if we have it in our file
pitch	Orientation pitch error and bound if we have it in our file
yaw	Orientation yaw error and bound if we have it in our file

13.34.3.4 calculate_nees()

```
void ResultTrajectory::calculate_nees (
            Statistics & nees_ori,
            Statistics & nees_pos )
```

Computes the Normalized Estimation Error Squared (NEES) for this trajectory.

If we have a covariance in addition to our pose estimate we can compute the NEES values. At each timestep we compute this for both orientation and position.

$$e_{nees,k} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i})^\top \mathbf{P}_{k,i}^{-1}(\mathbf{x}_{k,i} \boxminus \hat{\mathbf{x}}_{k,i})$$

Parameters

nees_ori	NEES values for the orientation
nees_pos	NEES values for the position

13.34.3.5 calculate_rpe()

```
void ResultTrajectory::calculate_rpe (
            const std::vector< double > & segment_lengths,
            std::map< double, std::pair< Statistics, Statistics > > & error_rpe )
```

Computes the Relative Pose Error (RPE) for this trajectory.

For the given set of segment lengths, this will split the trajectory into segments. From there it will compute the relative pose error for all segments. These are then returned as a map for each segment length.

$$\tilde{\mathbf{x}}_r = \mathbf{x}_k \boxminus \mathbf{x}_{k+d_i}$$
$$e_{rpe,d_i} = \frac{1}{D_i}\sum_{k=1}^{D_i} ||\tilde{\mathbf{x}}_r \boxminus \hat{\tilde{\mathbf{x}}}_r||_2^2$$

Parameters

segment_lengths	What segment lengths we want to calculate for
error_rpe	Map of segment lengths => errors for that length (orientation and position)

13.34.3.6 compute_comparison_indices_length()

```
std::vector< int > ov_eval::ResultTrajectory::compute_comparison_indices_length (
            std::vector< double > & distances,
            double distance,
            double max_dist_diff )  [inline], [protected]
```

Gets the indices at the end of subtractories of a given length when starting at each index. For each starting pose, find the end pose index which is the desired distance away.

Parameters

distances	Total distance travelled from start at each index
distance	Distance of subtrajectory
max_dist_diff	Maximum error between current trajectory length and the desired

Returns

End indices for each subtrajectory

13.35 ov_msckf::ROS1Visualizer Class Reference

Helper class that will publish results onto the ROS framework.

```
#include <ROS1Visualizer.h>
```

Public Member Functions

- ROS1Visualizer (std::shared_ptr< ros::NodeHandle > nh, std::shared_ptr< VioManager > app, std::shared_↩ptr< Simulator > sim=nullptr)

    Default constructor.
- void setup_subscribers (std::shared_ptr< ov_core::YamlParser > parser)

    Will setup ROS subscribers and callbacks.
- void **visualize** ()

    Will visualize the system if we have new things.
- void **visualize_odometry** (double timestamp)

Will publish our odometry message for the current timestep. This will take the current state estimate and get the propagated pose to the desired time. This can be used to get pose estimates on systems which require high frequency pose estimates.

- void **visualize_final** ()

  After the run has ended, print results.
- void **callback_inertial** (const sensor_msgs::Imu::ConstPtr &msg)

  Callback for inertial information.
- void **callback_monocular** (const sensor_msgs::ImageConstPtr &msg0, int cam_id0)

  Callback for monocular cameras information.
- void **callback_stereo** (const sensor_msgs::ImageConstPtr &msg0, const sensor_msgs::ImageConstPtr &msg1, int cam_id0, int cam_id1)

  Callback for synchronized stereo camera information.

Protected Types

- typedef message_filters::sync_policies::ApproximateTime< sensor_msgs::Image, sensor_msgs::Image > **sync_pol**

Protected Member Functions

- void **publish_state** ()

  Publish the current state.
- void **publish_images** ()

  Publish the active tracking image.
- void **publish_features** ()

  Publish current features.
- void **publish_groundtruth** ()

  Publish groundtruth (if we have it)
- void **publish_loopclosure_information** ()

  Publish loop-closure information of current pose and active track information.

Protected Attributes

- std::shared_ptr< ros::NodeHandle > **_nh**

  Global node handler.
- std::shared_ptr< VioManager > **_app**

  Core application of the filter system.
- std::shared_ptr< Simulator > **_sim**

  Simulator (is nullptr if we are not sim'ing)
- image_transport::Publisher **it_pub_tracks**
- image_transport::Publisher **it_pub_loop_img_depth**
- image_transport::Publisher **it_pub_loop_img_depth_color**
- ros::Publisher **pub_poseimu**
- ros::Publisher **pub_odomimu**
- ros::Publisher **pub_pathimu**
- ros::Publisher **pub_points_msckf**
- ros::Publisher **pub_points_slam**

- ros::Publisher **pub_points_aruco**
- ros::Publisher **pub_points_sim**
- ros::Publisher **pub_loop_pose**
- ros::Publisher **pub_loop_point**
- ros::Publisher **pub_loop_extrinsic**
- ros::Publisher **pub_loop_intrinsics**
- std::shared_ptr< tf::TransformBroadcaster > **mTfBr**
- ros::Subscriber **sub_imu**
- std::vector< ros::Subscriber > **subs_cam**
- std::vector< std::shared_ptr< message_filters::Synchronizer< sync_pol > > > **sync_cam**
- std::vector< std::shared_ptr< message_filters::Subscriber< sensor_msgs::Image > > > **sync_subs_cam**
- unsigned int **poses_seq_imu** = 0
- std::vector< geometry_msgs::PoseStamped > **poses_imu**
- ros::Publisher **pub_pathgt**
- ros::Publisher **pub_posegt**
- double **summed_rmse_ori** = 0.0
- double **summed_rmse_pos** = 0.0
- double **summed_nees_ori** = 0.0
- double **summed_nees_pos** = 0.0
- size_t **summed_number** = 0
- bool **start_time_set** = false
- boost::posix_time::ptime **rT1**
- boost::posix_time::ptime **rT2**
- double **last_visualization_timestamp** = 0
- std::map< double, Eigen::Matrix< double, 17, 1 > > **gt_states**
- unsigned int **poses_seq_gt** = 0
- std::vector< geometry_msgs::PoseStamped > **poses_gt**
- bool **publish_global2imu_tf** = true
- bool **publish_calibration_tf** = true
- bool **save_total_state**
- std::ofstream **of_state_est**
- std::ofstream **of_state_std**
- std::ofstream **of_state_gt**

13.35.1 Detailed Description

Helper class that will publish results onto the ROS framework.

Also save to file the current total state and covariance along with the groundtruth if we are simulating. We visualize the following things:

- State of the system on TF, pose message, and path
- Image of our tracker
- Our different features (SLAM, MSCKF, ARUCO)
- Groundtruth trajectory if we have it

13.35.2 Constructor & Destructor Documentation

13.35.2.1 ROS1Visualizer()

```
ROS1Visualizer::ROS1Visualizer (
            std::shared_ptr< ros::NodeHandle > nh,
            std::shared_ptr< VioManager > app,
            std::shared_ptr< Simulator > sim = nullptr )
```

Default constructor.

Parameters

nh	ROS node handler
app	Core estimator manager
sim	Simulator if we are simulating

13.35.3 Member Function Documentation

13.35.3.1 setup_subscribers()

```
void ROS1Visualizer::setup_subscribers (
            std::shared_ptr< ov_core::YamlParser > parser )
```

Will setup ROS subscribers and callbacks.

Parameters

parser	Configuration file parser

13.36 ov_msckf::ROS2Visualizer Class Reference

Helper class that will publish results onto the ROS framework.

```
#include <ROS2Visualizer.h>
```

Public Member Functions

- ROS2Visualizer (std::shared_ptr< rclcpp::Node > node, std::shared_ptr< VioManager > app, std::shared_ptr< Simulator > sim=nullptr)

  Default constructor.
- void setup_subscribers (std::shared_ptr< ov_core::YamlParser > parser)

  Will setup ROS subscribers and callbacks.
- void **visualize** ()

  Will visualize the system if we have new things.
- void **visualize_odometry** (double timestamp)

  Will publish our odometry message for the current timestep. This will take the current state estimate and get the propagated pose to the desired time. This can be used to get pose estimates on systems which require high frequency pose estimates.
- void **visualize_final** ()

  After the run has ended, print results.
- void **callback_inertial** (const sensor_msgs::msg::Imu::SharedPtr msg)

  Callback for inertial information.
- void **callback_monocular** (const sensor_msgs::msg::Image::SharedPtr msg0, int cam_id0)

  Callback for monocular cameras information.
- void **callback_stereo** (const sensor_msgs::msg::Image::ConstSharedPtr msg0, const sensor_msgs::msg::↩ Image::ConstSharedPtr msg1, int cam_id0, int cam_id1)

  Callback for synchronized stereo camera information.

Protected Types

- typedef message_filters::sync_policies::ApproximateTime< sensor_msgs::msg::Image, sensor_msgs::msg::↩ Image > **sync_pol**

Protected Member Functions

- void **publish_state** ()

  Publish the current state.
- void **publish_images** ()

  Publish the active tracking image.
- void **publish_features** ()

  Publish current features.
- void **publish_groundtruth** ()

  Publish groundtruth (if we have it)
- void **publish_loopclosure_information** ()

  Publish loop-closure information of current pose and active track information.

Protected Attributes

- std::shared_ptr< rclcpp::Node > **_node**

  Global node handler.
- std::shared_ptr< VioManager > **_app**

  Core application of the filter system.
- std::shared_ptr< Simulator > **_sim**

  Simulator (is nullptr if we are not sim'ing)
- image_transport::Publisher **it_pub_tracks**
- image_transport::Publisher **it_pub_loop_img_depth**
- image_transport::Publisher **it_pub_loop_img_depth_color**
- rclcpp::Publisher< geometry_msgs::msg::PoseWithCovarianceStamped >::SharedPtr **pub_poseimu**
- rclcpp::Publisher< nav_msgs::msg::Odometry >::SharedPtr **pub_odomimu**
- rclcpp::Publisher< nav_msgs::msg::Path >::SharedPtr **pub_pathimu**
- rclcpp::Publisher< sensor_msgs::msg::PointCloud2 >::SharedPtr **pub_points_msckf**
- rclcpp::Publisher< sensor_msgs::msg::PointCloud2 >::SharedPtr **pub_points_slam**
- rclcpp::Publisher< sensor_msgs::msg::PointCloud2 >::SharedPtr **pub_points_aruco**
- rclcpp::Publisher< sensor_msgs::msg::PointCloud2 >::SharedPtr **pub_points_sim**
- rclcpp::Publisher< nav_msgs::msg::Odometry >::SharedPtr **pub_loop_pose**
- rclcpp::Publisher< nav_msgs::msg::Odometry >::SharedPtr **pub_loop_extrinsic**
- rclcpp::Publisher< sensor_msgs::msg::PointCloud >::SharedPtr **pub_loop_point**
- rclcpp::Publisher< sensor_msgs::msg::CameraInfo >::SharedPtr **pub_loop_intrinsics**
- std::shared_ptr< tf2_ros::TransformBroadcaster > **mTfBr**
- rclcpp::Subscription< sensor_msgs::msg::Imu >::SharedPtr **sub_imu**
- std::vector< rclcpp::Subscription< sensor_msgs::msg::Image >::SharedPtr > **subs_cam**
- std::vector< std::shared_ptr< message_filters::Synchronizer< sync_pol > > > **sync_cam**
- std::vector< std::shared_ptr< message_filters::Subscriber< sensor_msgs::msg::Image > > > **sync_subs_↩ cam**
- std::vector< geometry_msgs::msg::PoseStamped > **poses_imu**
- rclcpp::Publisher< nav_msgs::msg::Path >::SharedPtr **pub_pathgt**
- rclcpp::Publisher< geometry_msgs::msg::PoseStamped >::SharedPtr **pub_posegt**
- double **summed_rmse_ori** = 0.0
- double **summed_rmse_pos** = 0.0
- double **summed_nees_ori** = 0.0
- double **summed_nees_pos** = 0.0
- size_t **summed_number** = 0
- bool **start_time_set** = false
- boost::posix_time::ptime **rT1**
- boost::posix_time::ptime **rT2**
- double **last_visualization_timestamp** = 0
- std::map< double, Eigen::Matrix< double, 17, 1 > > **gt_states**
- std::vector< geometry_msgs::msg::PoseStamped > **poses_gt**
- bool **publish_global2imu_tf** = true
- bool **publish_calibration_tf** = true
- bool **save_total_state**
- std::ofstream **of_state_est**
- std::ofstream **of_state_std**
- std::ofstream **of_state_gt**

13.36.1 Detailed Description

Helper class that will publish results onto the ROS framework.

Also save to file the current total state and covariance along with the groundtruth if we are simulating. We visualize the following things:

- State of the system on TF, pose message, and path
- Image of our tracker
- Our different features (SLAM, MSCKF, ARUCO)
- Groundtruth trajectory if we have it

13.36.2 Constructor & Destructor Documentation

13.36.2.1 ROS2Visualizer()

```
ROS2Visualizer::ROS2Visualizer (
            std::shared_ptr< rclcpp::Node > node,
            std::shared_ptr< VioManager > app,
            std::shared_ptr< Simulator > sim = nullptr )
```

Default constructor.

Parameters

node	ROS node pointer
app	Core estimator manager
sim	Simulator if we are simulating

13.36.3 Member Function Documentation

13.36.3.1 setup_subscribers()

```
void ROS2Visualizer::setup_subscribers (
            std::shared_ptr< ov_core::YamlParser > parser )
```

Will setup ROS subscribers and callbacks.

Parameters

parser	Configuration file parser

13.37 ov_msckf::RosVisualizerHelper Class Reference

Helper class that handles some common versions into and out of ROS formats.

```
#include <RosVisualizerHelper.h>
```

Static Public Member Functions

- static void sim_save_total_state_to_file (std::shared_ptr< State > state, std::shared_ptr< Simulator > sim, std↩ ::ofstream &of_state_est, std::ofstream &of_state_std, std::ofstream &of_state_gt)

  Save current estimate state and groundtruth including calibration.

13.37.1 Detailed Description

Helper class that handles some common versions into and out of ROS formats.

13.37.2 Member Function Documentation

13.37.2.1 sim_save_total_state_to_file()

```
static void ov_msckf::RosVisualizerHelper::sim_save_total_state_to_file (
            std::shared_ptr< State > state,
            std::shared_ptr< Simulator > sim,
            std::ofstream & of_state_est,
            std::ofstream & of_state_std,
            std::ofstream & of_state_gt )  [inline], [static]
```

Save current estimate state and groundtruth including calibration.

Parameters

state	Pointer to the state
sim	Pointer to the simulator (or null)
of_state_est	Output file for state estimate
of_state_std	Output file for covariance
of_state_gt	Output file for groundtruth (if we have it from sim)

13.38 ov_msckf::Simulator Class Reference

Master simulator class that generated visual-inertial measurements.

```
#include <Simulator.h>
```

Public Member Functions

- Simulator (VioManagerOptions &params_)

  Default constructor, will load all configuration variables.
- bool ok ()

  Returns if we are actively simulating.
- double current_timestamp ()

  Gets the timestamp we have simulated up too.
- bool get_state (double desired_time, Eigen::Matrix< double, 17, 1 > &imustate)

  Get the simulation state at a specified timestep.
- bool get_next_imu (double &time_imu, Eigen::Vector3d &wm, Eigen::Vector3d &am)

  Gets the next inertial reading if we have one.
- bool get_next_cam (double &time_cam, std::vector< int > &camids, std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > &feats)

  Gets the next inertial reading if we have one.
- std::unordered_map< size_t, Eigen::Vector3d > **get_map** ()

  Returns the true 3d map of features.
- std::vector< Eigen::Vector3d > **get_map_vec** ()

  Returns the true 3d map of features.
- VioManagerOptions **get_true_parameters** ()

  Access function to get the true parameters (i.e. calibration and settings)

Static Public Member Functions

- static void perturb_parameters (std::mt19937 gen_state, VioManagerOptions &params_)

  Will get a set of perturbed parameters.

Protected Member Functions

- std::vector< std::pair< size_t, Eigen::VectorXf > > project_pointcloud (const Eigen::Matrix3d &R_GtoI, const Eigen::Vector3d &p_IinG, int camid, const std::unordered_map< size_t, Eigen::Vector3d > &feats)

  Projects the passed map features into the desired camera frame.
- void generate_points (const Eigen::Matrix3d &R_GtoI, const Eigen::Vector3d &p_IinG, int camid, std::unordered↩ _map< size_t, Eigen::Vector3d > &feats, int numpts)

  Will generate points in the fov of the specified camera.

Protected Attributes

- VioManagerOptions **params**

    True vio manager params (a copy of the parsed ones)
- std::vector< Eigen::VectorXd > **traj_data**

    Our loaded trajectory data (timestamp(s), q_GtoI, p_IinG)
- ov_core::BsplineSE3 **spline**

    Our b-spline trajectory.
- size_t **id_map** = 0

    Our map of 3d features.
- std::unordered_map< size_t, Eigen::Vector3d > **featmap**
- std::mt19937 **gen_meas_imu**

    Mersenne twister PRNG for measurements (IMU)
- std::vector< std::mt19937 > **gen_meas_cams**

    Mersenne twister PRNG for measurements (CAMERAS)
- std::mt19937 **gen_state_init**

    Mersenne twister PRNG for state initialization.
- std::mt19937 **gen_state_perturb**

    Mersenne twister PRNG for state perturbations.
- bool **is_running**

    If our simulation is running.
- double **timestamp**

    Current timestamp of the system.
- double **timestamp_last_imu**

    Last time we had an IMU reading.
- double **timestamp_last_cam**

    Last time we had an CAMERA reading.
- Eigen::Vector3d **true_bias_accel** = Eigen::Vector3d::Zero()

    Our running acceleration bias.
- Eigen::Vector3d **true_bias_gyro** = Eigen::Vector3d::Zero()

    Our running gyroscope bias.
- std::vector< double > **hist_true_bias_time**
- std::vector< Eigen::Vector3d > **hist_true_bias_accel**
- std::vector< Eigen::Vector3d > **hist_true_bias_gyro**

13.38.1 Detailed Description

Master simulator class that generated visual-inertial measurements.

Given a trajectory this will generate a SE(3) ov_core::BsplineSE3 for that trajectory. This allows us to get the inertial measurement information at each timestep during this trajectory. After creating the bspline we will generate an environmental feature map which will be used as our feature measurements. This map will be projected into the frame at each timestep to get our "raw" uv measurements. We inject bias and white noises into our inertial readings while adding our white noise to the uv measurements also. The user should specify the sensor rates that they desire along with the seeds of the random number generators.

13.38.2 Constructor & Destructor Documentation

13.38.2.1 Simulator()

```
Simulator::Simulator (
            VioManagerOptions & params_ )
```

Default constructor, will load all configuration variables.

Parameters

params←_	VioManager parameters. Should have already been loaded from cmd.

13.38.3 Member Function Documentation

13.38.3.1 current_timestamp()

```
double ov_msckf::Simulator::current_timestamp ( )  [inline]
```

Gets the timestamp we have simulated up too.

Returns

Timestamp

13.38.3.2 generate_points()

```
void Simulator::generate_points (
            const Eigen::Matrix3d & R_GtoI,
            const Eigen::Vector3d & p_IinG,
            int camid,
            std::unordered_map< size_t, Eigen::Vector3d > & feats,
            int numpts )  [protected]
```

Will generate points in the fov of the specified camera.

Parameters

	R_GtoI	Orientation of the IMU pose
	p_IinG	Position of the IMU pose
	camid	Camera id of the camera sensor we want to project into
out	*feats*	Map we will append new features to
	numpts	Number of points we should generate

13.38.3.3 get_next_cam()

```
bool Simulator::get_next_cam (
            double & time_cam,
            std::vector< int > & camids,
            std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > & feats )
```

Gets the next inertial reading if we have one.

Parameters

time_cam	Time that this measurement occured at
camids	Camera ids that the corresponding vectors match
feats	Noisy uv measurements and ids for the returned time

Returns

True if we have a measurement

13.38.3.4 get_next_imu()

```
bool Simulator::get_next_imu (
            double & time_imu,
            Eigen::Vector3d & wm,
            Eigen::Vector3d & am )
```

Gets the next inertial reading if we have one.

Parameters

time_imu	Time that this measurement occured at
wm	Angular velocity measurement in the inertial frame
am	Linear velocity in the inertial frame

Returns

True if we have a measurement

13.38.3.5 get_state()

```
bool Simulator::get_state (
            double desired_time,
            Eigen::Matrix< double, 17, 1 > & imustate )
```

Get the simulation state at a specified timestep.

Parameters

desired_time	Timestamp we want to get the state at
imustate	State in the MSCKF ordering: [time(sec),q_GtoI,p_IinG,v_IinG,b_gyro,b_accel]

Returns

True if we have a state

13.38.3.6 ok()

```
bool ov_msckf::Simulator::ok ( ) [inline]
```

Returns if we are actively simulating.

Returns

True if we still have simulation data

13.38.3.7 perturb_parameters()

```
void Simulator::perturb_parameters (
            std::mt19937 gen_state,
            VioManagerOptions & params_ ) [static]
```

Will get a set of perturbed parameters.

Parameters

gen_state	Random number gen to use
params↩ *_*	Parameters we will perturb

13.38.3.8 project_pointcloud()

```
std::vector< std::pair< size_t, Eigen::VectorXf > > Simulator::project_pointcloud (
             const Eigen::Matrix3d & R_GtoI,
             const Eigen::Vector3d & p_IinG,
             int camid,
             const std::unordered_map< size_t, Eigen::Vector3d > & feats )  [protected]
```

Projects the passed map features into the desired camera frame.

Parameters

R_GtoI	Orientation of the IMU pose
p_IinG	Position of the IMU pose
camid	Camera id of the camera sensor we want to project into
feats	Our set of 3d features

Returns

True distorted raw image measurements and their ids for the specified camera

13.39 ov_msckf::State Class Reference

State of our filter.

```
#include <State.h>
```

Public Member Functions

- State (StateOptions &options_)

  Default Constructor (will initialize variables to defaults)
- double margtimestep ()

  Will return the timestep that we will marginalize next. As of right now, since we are using a sliding window, this is the oldest clone. But if you wanted to do a keyframe system, you could selectively marginalize clones.
- int max_covariance_size ()

  Calculates the current max size of the covariance.

Public Attributes

- double **_timestamp** = -1

  Current timestamp (should be the last update time!)
- StateOptions **_options**

  Struct containing filter options.
- std::shared_ptr< ov_type::IMU > **_imu**

  Pointer to the "active" IMU state (q_GtoI, p_IinG, v_IinG, bg, ba)
- std::map< double, std::shared_ptr< ov_type::PoseJPL > > **_clones_IMU**

  Map between imaging times and clone poses (q_GtoIi, p_IiinG)
- std::unordered_map< size_t, std::shared_ptr< ov_type::Landmark > > **_features_SLAM**

  Our current set of SLAM features (3d positions)
- std::shared_ptr< ov_type::Vec > **_calib_dt_CAMtoIMU**

  Time offset base IMU to camera (t_imu = t_cam + t_off)
- std::unordered_map< size_t, std::shared_ptr< ov_type::PoseJPL > > **_calib_IMUtoCAM**

  Calibration poses for each camera (R_ItoC, p_IinC)
- std::unordered_map< size_t, std::shared_ptr< ov_type::Vec > > **_cam_intrinsics**

  Camera intrinsics.
- std::unordered_map< size_t, std::shared_ptr< ov_core::CamBase > > **_cam_intrinsics_cameras**

  Camera intrinsics camera objects.

Friends

- class **StateHelper**

13.39.1 Detailed Description

State of our filter.

This state has all the current estimates for the filter. This system is modeled after the MSCKF filter, thus we have a sliding window of clones. We additionally have more parameters for online estimation of calibration and SLAM features. We also have the covariance of the system, which should be managed using the StateHelper class.

13.39.2 Constructor & Destructor Documentation

13.39.2.1 State()

```
State::State (
            StateOptions & options_ )
```

Default Constructor (will initialize variables to defaults)

Parameters

options←↩ _	Options structure containing filter options

13.39.3 Member Function Documentation

13.39.3.1 margtimestep()

```
double ov_msckf::State::margtimestep ( )  [inline]
```

Will return the timestep that we will marginalize next. As of right now, since we are using a sliding window, this is the oldest clone. But if you wanted to do a keyframe system, you could selectively marginalize clones.

Returns

timestep of clone we will marginalize

13.39.3.2 max_covariance_size()

```
int ov_msckf::State::max_covariance_size ( )  [inline]
```

Calculates the current max size of the covariance.

Returns

Size of the current covariance matrix

13.40 ov_msckf::StateHelper Class Reference

Helper which manipulates the State and its covariance.

```
#include <StateHelper.h>
```

Static Public Member Functions

- static void EKFPropagation (std::shared_ptr< State > state, const std::vector< std::shared_ptr< ov_type::Type > > &order_NEW, const std::vector< std::shared_ptr< ov_type::Type > > &order_OLD, const Eigen::MatrixXd &Phi, const Eigen::MatrixXd &Q)

  Performs EKF propagation of the state covariance.
- static void EKFUpdate (std::shared_ptr< State > state, const std::vector< std::shared_ptr< ov_type::Type > > &H_order, const Eigen::MatrixXd &H, const Eigen::VectorXd &res, const Eigen::MatrixXd &R)

  Performs EKF update of the state (see Linear Measurement Update page)
- static void set_initial_covariance (std::shared_ptr< State > state, const Eigen::MatrixXd &covariance, const std↩ ::vector< std::shared_ptr< ov_type::Type > > &order)

  This will set the initial covaraince of the specified state elements. Will also ensure that proper cross-covariances are inserted.
- static Eigen::MatrixXd get_marginal_covariance (std::shared_ptr< State > state, const std::vector< std↩ ::shared_ptr< ov_type::Type > > &small_variables)

  For a given set of variables, this will this will calculate a smaller covariance.
- static Eigen::MatrixXd get_full_covariance (std::shared_ptr< State > state)

  This gets the full covariance matrix.
- static void marginalize (std::shared_ptr< State > state, std::shared_ptr< ov_type::Type > marg)

  Marginalizes a variable, properly modifying the ordering/covariances in the state.
- static std::shared_ptr< ov_type::Type > clone (std::shared_ptr< State > state, std::shared_ptr< ov_type::Type > variable_to_clone)

  Clones "variable to clone" and places it at end of covariance.
- static bool initialize (std::shared_ptr< State > state, std::shared_ptr< ov_type::Type > new_variable, const std↩ ::vector< std::shared_ptr< ov_type::Type > > &H_order, Eigen::MatrixXd &H_R, Eigen::MatrixXd &H_L, Eigen↩ ::MatrixXd &R, Eigen::VectorXd &res, double chi_2_mult)

  Initializes new variable into covariance.
- static void initialize_invertible (std::shared_ptr< State > state, std::shared_ptr< ov_type::Type > new_variable, const std::vector< std::shared_ptr< ov_type::Type > > &H_order, const Eigen::MatrixXd &H_R, const Eigen::↩ MatrixXd &H_L, const Eigen::MatrixXd &R, const Eigen::VectorXd &res)

  Initializes new variable into covariance (H_L must be invertible)
- static void augment_clone (std::shared_ptr< State > state, Eigen::Matrix< double, 3, 1 > last_w)

  Augment the state with a stochastic copy of the current IMU pose.
- static void marginalize_old_clone (std::shared_ptr< State > state)

  Remove the oldest clone, if we have more then the max clone count!!
- static void marginalize_slam (std::shared_ptr< State > state)

  Marginalize bad SLAM features.

13.40.1 Detailed Description

Helper which manipulates the State and its covariance.

In general, this class has all the core logic for an Extended Kalman Filter (EKF)-based system. This has all functions that change the covariance along with addition and removing elements from the state. All functions here are static, and thus are self-contained so that in the future multiple states could be tracked and updated. We recommend you look directly at the code for this class for clarity on what exactly we are doing in each and the matching documentation pages.

13.40.2 Member Function Documentation

13.40.2.1 augment_clone()

```
void StateHelper::augment_clone (
            std::shared_ptr< State > state,
            Eigen::Matrix< double, 3, 1 > last_w )  [static]
```

Augment the state with a stochastic copy of the current IMU pose.

After propagation, normally we augment the state with an new clone that is at the new update timestep. This augmentation clones the IMU pose and adds it to our state's clone map. If we are doing time offset calibration we also make our cloning a function of the time offset. Time offset logic is based on Mingyang Li and Anastasios I. Mourikis paper: http://journals.sagepub.com/doi/pdf/10.1177/0278364913515286 We can write the current clone at the true imu base clock time as the follow:

$$
{}^{I_{t+t_d}}_{G}\bar{q} = \begin{bmatrix} \frac{1}{2}\,{}^{I_{t+\hat{t}_d}}\boldsymbol{\omega}\tilde{t}_d \\ 1 \end{bmatrix} \otimes {}^{I_{t+\hat{t}_d}}_{G}\bar{q}
$$
$$
{}^{G}\mathbf{p}_{I_{t+t_d}} = {}^{G}\mathbf{p}_{I_{t+\hat{t}_d}} + {}^{G}\mathbf{v}_{I_{t+\hat{t}_d}}\tilde{t}_d
$$

where we say that we have propagated our state up to the current estimated true imaging time for the current image, ${}^{I_{t+\hat{t}_d}}\boldsymbol{\omega}$ is the angular velocity at the end of propagation with biases removed. This is off by some smaller error, so to get to the true imaging time in the imu base clock, we can append some small timeoffset error. Thus the Jacobian in respect to our time offset during our cloning procedure is the following:

$$
\frac{\partial\, {}^{I_{t+t_d}}_{G}\tilde{\boldsymbol{\theta}}}{\partial \tilde{t}_d} = {}^{I_{t+\hat{t}_d}}\boldsymbol{\omega}
$$
$$
\frac{\partial\, {}^{G}\tilde{\mathbf{p}}_{I_{t+t_d}}}{\partial \tilde{t}_d} = {}^{G}\mathbf{v}_{I_{t+\hat{t}_d}}
$$

Parameters

state	Pointer to state
last←↩ _w	The estimated angular velocity at cloning time (used to estimate imu-cam time offset)

13.40.2.2 clone()

```
std::shared_ptr< Type > StateHelper::clone (
            std::shared_ptr< State > state,
            std::shared_ptr< ov_type::Type > variable_to_clone )  [static]
```

Clones "variable to clone" and places it at end of covariance.

Parameters

state	Pointer to state
variable_to_clone	Pointer to variable that will be cloned

13.40.2.3 EKFPropagation()

```
void StateHelper::EKFPropagation (
            std::shared_ptr< State > state,
            const std::vector< std::shared_ptr< ov_type::Type > > & order_NEW,
            const std::vector< std::shared_ptr< ov_type::Type > > & order_OLD,
            const Eigen::MatrixXd & Phi,
            const Eigen::MatrixXd & Q )  [static]
```

Performs EKF propagation of the state covariance.

The mean of the state should already have been propagated, thus just moves the covariance forward in time. The new states that we are propagating the old covariance into, should be **contiguous** in memory. The user only needs to specify the sub-variables that this block is a function of.

$$\tilde{\mathbf{x}}' = \begin{bmatrix} \mathbf{\Phi}_1 & \mathbf{\Phi}_2 & \mathbf{\Phi}_3 \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}}_1 \\ \tilde{\mathbf{x}}_2 \\ \tilde{\mathbf{x}}_3 \end{bmatrix} + \mathbf{n}$$

Parameters

state	Pointer to state
order_NEW	Contiguous variables that have evolved according to this state transition
order_OLD	Variable ordering used in the state transition
Phi	State transition matrix (size order_NEW by size order_OLD)
Q	Additive state propagation noise matrix (size order_NEW by size order_NEW)

13.40.2.4 EKFUpdate()

```
void StateHelper::EKFUpdate (
            std::shared_ptr< State > state,
            const std::vector< std::shared_ptr< ov_type::Type > > & H_order,
            const Eigen::MatrixXd & H,
            const Eigen::VectorXd & res,
            const Eigen::MatrixXd & R )  [static]
```

Performs EKF update of the state (see Linear Measurement Update page)

Parameters

state	Pointer to state
H_order	Variable ordering used in the compressed Jacobian
H	Condensed Jacobian of updating measurement
res	residual of updating measurement
R	updating measurement covariance

13.40.2.5 get_full_covariance()

```
Eigen::MatrixXd StateHelper::get_full_covariance (
            std::shared_ptr< State > state )  [static]
```

This gets the full covariance matrix.

Should only be used during simulation as operations on this covariance will be slow. This will return a copy, so this cannot be used to change the covariance by design. Please use the other interface functions in the StateHelper to progamatically change to covariance.

Parameters

state	Pointer to state

Returns

covariance of current state

13.40.2.6 get_marginal_covariance()

```
Eigen::MatrixXd StateHelper::get_marginal_covariance (
            std::shared_ptr< State > state,
            const std::vector< std::shared_ptr< ov_type::Type > > & small_variables )  [static]
```

For a given set of variables, this will this will calculate a smaller covariance.

That only includes the ones specified with all crossterms. Thus the size of the return will be the summed dimension of all the passed variables. Normal use for this is a chi-squared check before update (where you don't need the full covariance).

Parameters

state	Pointer to state
small_variables	Vector of variables whose marginal covariance is desired

Returns

marginal covariance of the passed variables

13.40.2.7 initialize()

```
bool StateHelper::initialize (
             std::shared_ptr< State > state,
             std::shared_ptr< ov_type::Type > new_variable,
             const std::vector< std::shared_ptr< ov_type::Type > > & H_order,
             Eigen::MatrixXd & H_R,
             Eigen::MatrixXd & H_L,
             Eigen::MatrixXd & R,
             Eigen::VectorXd & res,
             double chi_2_mult )  [static]
```

Initializes new variable into covariance.

Uses Givens to separate into updating and initializing systems (therefore system must be fed as isotropic). If you are not isotropic first whiten your system (TODO: we should add a helper function to do this). If your H_L Jacobian is already directly invertable, the just call the initialize_invertible() instead of this function. Please refer to Delayed Feature Initialization page for detailed derivation.

Parameters

state	Pointer to state
new_variable	Pointer to variable to be initialized
H_order	Vector of pointers in order they are contained in the condensed state Jacobian
H_R	Jacobian of initializing measurements wrt variables in H_order
H_L	Jacobian of initializing measurements wrt new variable
R	Covariance of initializing measurements (isotropic)
res	Residual of initializing measurements
chi_2_mult	Value we should multiply the chi2 threshold by (larger means it will be accepted more measurements)

13.40.2.8 initialize_invertible()

```
void StateHelper::initialize_invertible (
             std::shared_ptr< State > state,
             std::shared_ptr< ov_type::Type > new_variable,
             const std::vector< std::shared_ptr< ov_type::Type > > & H_order,
             const Eigen::MatrixXd & H_R,
             const Eigen::MatrixXd & H_L,
             const Eigen::MatrixXd & R,
             const Eigen::VectorXd & res )  [static]
```

Initializes new variable into covariance (H_L must be invertible)

Please refer to Delayed Feature Initialization page for detailed derivation. This is just the update assuming that H_L is invertable (and thus square) and isotropic noise.

Parameters

state	Pointer to state
new_variable	Pointer to variable to be initialized
H_order	Vector of pointers in order they are contained in the condensed state Jacobian
H_R	Jacobian of initializing measurements wrt variables in H_order
H_L	Jacobian of initializing measurements wrt new variable (needs to be invertible)
R	Covariance of initializing measurements
res	Residual of initializing measurements

13.40.2.9 marginalize()

```
void StateHelper::marginalize (
            std::shared_ptr< State > state,
            std::shared_ptr< ov_type::Type > marg )  [static]
```

Marginalizes a variable, properly modifying the ordering/covariances in the state.

This function can support any Type variable out of the box. Right now the marginalization of a sub-variable/type is not supported. For example if you wanted to just marginalize the orientation of a PoseJPL, that isn't supported. We will first remove the rows and columns corresponding to the type (i.e. do the marginalization). After we update all the type ids so that they take into account that the covariance has shrunk in parts of it.

Parameters

state	Pointer to state
marg	Pointer to variable to marginalize

13.40.2.10 marginalize_old_clone()

```
static void ov_msckf::StateHelper::marginalize_old_clone (
            std::shared_ptr< State > state )  [inline], [static]
```

Remove the oldest clone, if we have more then the max clone count!!

This will marginalize the clone from our covariance, and remove it from our state. This is mainly a helper function that we can call after each update. It will marginalize the clone specified by State::margtimestep() which should return a clone timestamp.

Parameters

state	Pointer to state

13.40.2.11 marginalize_slam()

```
static void ov_msckf::StateHelper::marginalize_slam (
            std::shared_ptr< State > state )  [inline], [static]
```

Marginalize bad SLAM features.

Parameters

state	Pointer to state

13.40.2.12 set_initial_covariance()

```
void StateHelper::set_initial_covariance (
            std::shared_ptr< State > state,
            const Eigen::MatrixXd & covariance,
            const std::vector< std::shared_ptr< ov_type::Type > > & order )  [static]
```

This will set the initial covaraince of the specified state elements. Will also ensure that proper cross-covariances are inserted.

Parameters

state	Pointer to state
covariance	The covariance of the system state
order	Order of the covariance matrix

13.41 ov_msckf::StateOptions Struct Reference

Struct which stores all our filter options.

`#include <StateOptions.h>`

Public Member Functions

- void **print** (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  Nice print function of what parameters we have loaded.

Public Attributes

- bool **do_fej** = true

  Bool to determine whether or not to do first estimate Jacobians.
- bool **imu_avg** = false

  Bool to determine whether or not use imu message averaging.
- bool **use_rk4_integration** = true

  Bool to determine if we should use Rk4 imu integration.
- bool **do_calib_camera_pose** = false

  Bool to determine whether or not to calibrate imu-to-camera pose.
- bool **do_calib_camera_intrinsics** = false

  Bool to determine whether or not to calibrate camera intrinsics.
- bool **do_calib_camera_timeoffset** = false

  Bool to determine whether or not to calibrate camera to IMU time offset.
- int **max_clone_size** = 11

  Max clone size of sliding window.
- int **max_slam_features** = 25

  Max number of estimated SLAM features.
- int **max_slam_in_update** = 1000

  Max number of SLAM features we allow to be included in a single EKF update.
- int **max_msckf_in_update** = 1000

  Max number of MSCKF features we will use at a given image timestep.
- int **max_aruco_features** = 1024

  Max number of estimated ARUCO features.
- int **num_cameras** = 1

  Number of distinct cameras that we will observe features in.
- ov_type::LandmarkRepresentation::Representation **feat_rep_msckf** = ov_type::LandmarkRepresentation::↩ Representation::GLOBAL_3D

  What representation our features are in (msckf features)
- ov_type::LandmarkRepresentation::Representation **feat_rep_slam** = ov_type::LandmarkRepresentation::↩ Representation::GLOBAL_3D

  What representation our features are in (slam features)
- ov_type::LandmarkRepresentation::Representation **feat_rep_aruco** = ov_type::LandmarkRepresentation::↩ Representation::GLOBAL_3D

  What representation our features are in (aruco tag features)

13.41.1 Detailed Description

Struct which stores all our filter options.

13.42 ov_init::StaticInitializer Class Reference

Initializer for a static visual-inertial system.

```
#include <StaticInitializer.h>
```

Public Member Functions

- StaticInitializer (InertialInitializerOptions &params_, std::shared_ptr< ov_core::FeatureDatabase > db, std↩ ::shared_ptr< std::vector< ov_core::ImuData > > imu_data_)

  Default constructor.
- bool initialize (double &timestamp, Eigen::MatrixXd &covariance, std::vector< std::shared_ptr< ov_type::Type > > &order, std::shared_ptr< ov_type::IMU > t_imu, bool wait_for_jerk=true)

  Try to get the initialized system using just the imu.

13.42.1 Detailed Description

Initializer for a static visual-inertial system.

This implementation that assumes that the imu starts from standing still.

To initialize from standstill:

1. Collect all inertial measurements
2. See if within the last window there was a jump in acceleration
3. If the jump is past our threshold we should init (i.e. we have started moving)
4. Use the *previous* window, which should have been stationary to initialize orientation
5. Return a roll and pitch aligned with gravity and biases.

13.42.2 Constructor & Destructor Documentation

13.42.2.1 StaticInitializer()

```
ov_init::StaticInitializer::StaticInitializer (
            InertialInitializerOptions & params_,
            std::shared_ptr< ov_core::FeatureDatabase > db,
            std::shared_ptr< std::vector< ov_core::ImuData > > imu_data_ )  [inline], [explicit]
```

Default constructor.

Parameters

params↩ _	Parameters loaded from either ROS or CMDLINE
db	Feature tracker database with all features in it
*imu_↩ data_*	Shared pointer to our IMU vector of historical information

13.42.3 Member Function Documentation

13.42.3.1 initialize()

```
bool StaticInitializer::initialize (
            double & timestamp,
            Eigen::MatrixXd & covariance,
            std::vector< std::shared_ptr< ov_type::Type > > & order,
            std::shared_ptr< ov_type::IMU > t_imu,
            bool wait_for_jerk = true )
```

Try to get the initialized system using just the imu.

This will check if we have had a large enough jump in our acceleration. If we have then we will use the period of time before this jump to initialize the state. This assumes that our imu is sitting still and is not moving (so this would fail if we are experiencing constant acceleration).

In the case that we do not wait for a jump (i.e. `wait_for_jerk` is false), then the system will try to initialize as soon as possible. This is only recommended if you have zero velocity update enabled to handle the stationary cases. To initialize in this case, we need to have the average angular variance be below the set threshold (i.e. we need to be stationary).

Parameters

out	*timestamp*	Timestamp we have initialized the state at
out	*covariance*	Calculated covariance of the returned state
out	*order*	Order of the covariance matrix
out	*t_imu*	Our imu type element
	wait_for_jerk	If true we will wait for a "jerk"

Returns

True if we have successfully initialized our system

13.43 ov_eval::Statistics Struct Reference

Statistics object for a given set scalar time series values.

`#include <Statistics.h>`

Public Member Functions

- void **calculate** ()

  Will calculate all values from our vectors of information.
- void **clear** ()

  Will clear any old values.

Public Attributes

- double **rmse** = 0.0

  Root mean squared for the given values.
- double **mean** = 0.0

  Mean of the given values.
- double **median** = 0.0

  Median of the given values.
- double **std** = 0.0

  Standard deviation of given values.
- double **max** = 0.0

  Max of the given values.
- double **min** = 0.0

  Min of the given values.
- double **ninetynine** = 0.0

  99th percentile
- std::vector< double > **timestamps**

  Timestamp when these values occured at.
- std::vector< double > **values**

  Values (e.g. error or nees at a given time)
- std::vector< double > **values_bound**

  Bound of these values (e.g. our expected covariance bound)

13.43.1 Detailed Description

Statistics object for a given set scalar time series values.

Ensure that you call the calculate() function to update the values before using them. This will compute all the final results from the values in values vector.

13.44 ov_core::TrackAruco Class Reference

Tracking of OpenCV Aruoc tags.

```
#include <TrackAruco.h>
```

Public Member Functions

- TrackAruco (std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras, int numaruco, bool binocular, HistogramMethod histmethod, bool downsize)

  Public constructor with configuration variables.
- void feed_new_camera (const CameraData &message)

  Process a new image.

Protected Attributes

- int **max_tag_id**
- bool **do_downsizing**

Additional Inherited Members

13.44.1 Detailed Description

Tracking of OpenCV Aruoc tags.

This class handles the tracking of `OpenCV Aruco tags`. We track the corners of the tag as compared to the pose of the tag or any other corners. Right now we hardcode the dictionary to be `cv::aruco::DICT_6X6_1000`, so please generate tags in this family of tags. You can generate these tags using an online utility: `https://chev.↩ me/arucogen/` The actual size of the tags do not matter since we do not recover the pose and instead just use this for re-detection and tracking of the four corners of the tag.

13.44.2 Constructor & Destructor Documentation

13.44.2.1 TrackAruco()

```
ov_core::TrackAruco::TrackAruco (
            std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras,
            int numaruco,
            bool binocular,
            HistogramMethod histmethod,
            bool downsize )  [inline], [explicit]
```

Public constructor with configuration variables.

Parameters

cameras	camera calibration object which has all camera intrinsics in it
numaruco	the max id of the arucotags, we don't use any tags greater than this value even if we extract them
binocular	if we should do binocular feature tracking or stereo if there are multiple cameras
histmethod	what type of histogram pre-processing should be done (histogram eq?)
downsize	we can scale the image by 1/2 to increase Aruco tag extraction speed

13.44.3 Member Function Documentation

13.44.3.1 feed_new_camera()

```
void TrackAruco::feed_new_camera (
            const CameraData & message )  [virtual]
```

Process a new image.

Parameters

message	Contains our timestamp, images, and camera ids

Implements ov_core::TrackBase.

13.45 ov_core::TrackBase Class Reference

Visual feature tracking base class.

```
#include <TrackBase.h>
```

Public Types

- enum HistogramMethod { **NONE** , **HISTOGRAM** , **CLAHE** }

  Desired pre-processing image method.

Public Member Functions

- TrackBase (std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras, int numfeats, int numaruco, bool stereo, HistogramMethod histmethod)

  Public constructor with configuration variables.
- virtual void feed_new_camera (const CameraData &message)=0

  Process a new image.
- virtual void display_active (cv::Mat &img_out, int r1, int g1, int b1, int r2, int g2, int b2)

  Shows features extracted in the last image.
- virtual void display_history (cv::Mat &img_out, int r1, int g1, int b1, int r2, int g2, int b2, std::vector< size_t > highlighted={})

  Shows a "trail" for each feature (i.e. its history)
- std::shared_ptr< FeatureDatabase > get_feature_database ()

  Get the feature database with all the track information.
- void change_feat_id (size_t id_old, size_t id_new)

  Changes the ID of an actively tracked feature to another one.
- int **get_num_features** ()

  Getter method for number of active features.
- void **set_num_features** (int _num_features)

  Setter method for number of active features.

Protected Attributes

- std::unordered_map< size_t, std::shared_ptr< CamBase > > **camera_calib**

  Camera object which has all calibration in it.
- std::shared_ptr< FeatureDatabase > **database**

  Database with all our current features.
- std::map< size_t, bool > **camera_fisheye**

  If we are a fisheye model or not.
- int **num_features**

  Number of features we should try to track frame to frame.
- bool **use_stereo**

  If we should use binocular tracking or stereo tracking for multi-camera.
- HistogramMethod **histogram_method**

  What histogram equalization method we should pre-process images with?
- std::vector< std::mutex > **mtx_feeds**

  Mutexs for our last set of image storage (img_last, pts_last, and ids_last)
- std::map< size_t, cv::Mat > **img_last**

  Last set of images (use map so all trackers render in the same order)
- std::map< size_t, cv::Mat > **img_mask_last**

  Last set of images (use map so all trackers render in the same order)
- std::unordered_map< size_t, std::vector< cv::KeyPoint > > **pts_last**

  Last set of tracked points.
- std::unordered_map< size_t, std::vector< size_t > > **ids_last**

  Set of IDs of each current feature in the database.
- std::atomic< size_t > **currid**

  Master ID for this tracker (atomic to allow for multi-threading)
- boost::posix_time::ptime **rT1**
- boost::posix_time::ptime **rT2**
- boost::posix_time::ptime **rT3**
- boost::posix_time::ptime **rT4**
- boost::posix_time::ptime **rT5**
- boost::posix_time::ptime **rT6**
- boost::posix_time::ptime **rT7**

13.45.1 Detailed Description

Visual feature tracking base class.

This is the base class for all our visual trackers. The goal here is to provide a common interface so all underlying trackers can simply hide away all the complexities. We have something called the "feature database" which has all the tracking information inside of it. The user can ask this database for features which can then be used in an MSCKF or batch-based setting. The feature tracks store both the raw (distorted) and undistorted/normalized values. Right now we just support two camera models, see: undistort_point_brown() and undistort_point_fisheye().

A Note on Multi-Threading Support

> There is some support for asynchronous multi-threaded feature tracking of independent cameras. The key assumption during implementation is that the user will not try to track on the same camera in parallel, and instead call on different cameras. For example, if I have two cameras, I can either sequentially call the feed function, or I spin each of these into separate threads and wait for their return. The currid is atomic to allow for multiple threads to access it without issue and ensure that all features have unique id values. We also have mutex for access for the calibration and previous images and tracks (used during visualization). It should be noted that if a thread calls visualization, it might hang or the feed thread might, due to acquiring the mutex for that specific camera id / feed.

This base class also handles most of the heavy lifting with the visualization, but the sub-classes can override this and do their own logic if they want (i.e. the TrackAruco has its own logic for visualization). This visualization needs access to the prior images and their tracks, thus must synchronise in the case of multi-threading. This shouldn't impact performance, but high frequency visualization calls can negatively effect the performance.

13.45.2 Constructor & Destructor Documentation

13.45.2.1 TrackBase()

```
ov_core::TrackBase::TrackBase (
            std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras,
            int numfeats,
            int numaruco,
            bool stereo,
            HistogramMethod histmethod )  [inline]
```

Public constructor with configuration variables.

Parameters

cameras	camera calibration object which has all camera intrinsics in it
numfeats	number of features we want want to track (i.e. track 200 points from frame to frame)
numaruco	the max id of the arucotags, so we ensure that we start our non-auroc features above this value
stereo	if we should do stereo feature tracking or binocular
histmethod	what type of histogram pre-processing should be done (histogram eq?)

13.45.3 Member Function Documentation

13.45.3.1 change_feat_id()

```
void ov_core::TrackBase::change_feat_id (
            size_t id_old,
            size_t id_new )  [inline]
```

Changes the ID of an actively tracked feature to another one.

This function can be helpfull if you detect a loop-closure with an old frame. One could then change the id of an active feature to match the old feature id!

Parameters

id_old	Old id we want to change
id_new	Id we want to change the old id to

13.45.3.2 display_active()

```
void TrackBase::display_active (
            cv::Mat & img_out,
            int r1,
            int g1,
            int b1,
            int r2,
            int g2,
            int b2 )  [virtual]
```

Shows features extracted in the last image.

Parameters

img_out	image to which we will overlayed features on
r1,g1,b1	first color to draw in
r2,g2,b2	second color to draw in

13.45.3.3 display_history()

```
void TrackBase::display_history (
            cv::Mat & img_out,
            int r1,
            int g1,
            int b1,
            int r2,
            int g2,
            int b2,
            std::vector< size_t > highlighted = {} )  [virtual]
```

Shows a "trail" for each feature (i.e. its history)

Parameters

img_out	image to which we will overlayed features on
r1,g1,b1	first color to draw in
r2,g2,b2	second color to draw in
highlighted	unique ids which we wish to highlight (e.g. slam feats)

13.45.3.4 feed_new_camera()

```
virtual void ov_core::TrackBase::feed_new_camera (
            const CameraData & message )  [pure virtual]
```

Process a new image.

Parameters

message	Contains our timestamp, images, and camera ids

Implemented in ov_core::TrackAruco, ov_core::TrackDescriptor, ov_core::TrackKLT, and ov_core::TrackSIM.

13.45.3.5 get_feature_database()

```
std::shared_ptr< FeatureDatabase > ov_core::TrackBase::get_feature_database ( )  [inline]
```

Get the feature database with all the track information.

Returns

FeatureDatabase pointer that one can query for features

13.46 ov_core::TrackDescriptor Class Reference

Descriptor-based visual tracking.

```
#include <TrackDescriptor.h>
```

Public Member Functions

- TrackDescriptor (std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras, int numfeats, int numaruco, bool binocular, HistogramMethod histmethod, int fast_threshold, int gridx, int gridy, int minpxdist, double knnratio)

    Public constructor with configuration variables.
- void feed_new_camera (const CameraData &message)

    Process a new image.

Protected Member Functions

- void feed_monocular (const CameraData &message, size_t msg_id)

    Process a new monocular image.
- void feed_stereo (const CameraData &message, size_t msg_id_left, size_t msg_id_right)

    Process new stereo pair of images.
- void perform_detection_monocular (const cv::Mat &img0, const cv::Mat &mask0, std::vector< cv::KeyPoint > &pts0, cv::Mat &desc0, std::vector< size_t > &ids0)

    Detects new features in the current image.
- void perform_detection_stereo (const cv::Mat &img0, const cv::Mat &img1, const cv::Mat &mask0, const cv::Mat &mask1, std::vector< cv::KeyPoint > &pts0, std::vector< cv::KeyPoint > &pts1, cv::Mat &desc0, cv::Mat &desc1, size_t cam_id0, size_t cam_id1, std::vector< size_t > &ids0, std::vector< size_t > &ids1)

    Detects new features in the current stereo pair.
- void robust_match (std::vector< cv::KeyPoint > &pts0, std::vector< cv::KeyPoint > pts1, cv::Mat &desc0, cv::Mat &desc1, size_t id0, size_t id1, std::vector< cv::DMatch > &matches)

    Find matches between two keypoint+descriptor sets.
- void **robust_ratio_test** (std::vector< std::vector< cv::DMatch > > &matches)
- void **robust_symmetry_test** (std::vector< std::vector< cv::DMatch > > &matches1, std::vector< std::vector< cv::DMatch > > &matches2, std::vector< cv::DMatch > &good_matches)

Protected Attributes

- boost::posix_time::ptime **rT1**
- boost::posix_time::ptime **rT2**
- boost::posix_time::ptime **rT3**
- boost::posix_time::ptime **rT4**
- boost::posix_time::ptime **rT5**
- boost::posix_time::ptime **rT6**
- boost::posix_time::ptime **rT7**
- cv::Ptr< cv::ORB > **orb0** = cv::ORB::create()
- cv::Ptr< cv::ORB > **orb1** = cv::ORB::create()
- cv::Ptr< cv::DescriptorMatcher > **matcher** = cv::DescriptorMatcher::create("BruteForce-Hamming")
- int **threshold**
- int **grid_x**
- int **grid_y**
- int **min_px_dist**
- double **knn_ratio**
- std::unordered_map< size_t, cv::Mat > **desc_last**

Additional Inherited Members

13.46.1 Detailed Description

Descriptor-based visual tracking.

Here we use descriptor matching to track features from one frame to the next. We track both temporally, and across stereo pairs to get stereo constraints. Right now we use ORB descriptors as we have found it is the fastest when computing descriptors. Tracks are then rejected based on a ratio test and ransac.

13.46.2 Constructor & Destructor Documentation

13.46.2.1 TrackDescriptor()

```
ov_core::TrackDescriptor::TrackDescriptor (
             std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras,
             int numfeats,
             int numaruco,
             bool binocular,
             HistogramMethod histmethod,
             int fast_threshold,
             int gridx,
             int gridy,
             int minpxdist,
             double knnratio )  [inline], [explicit]
```

Public constructor with configuration variables.

Parameters

cameras	camera calibration object which has all camera intrinsics in it
numfeats	number of features we want want to track (i.e. track 200 points from frame to frame)
numaruco	the max id of the arucotags, so we ensure that we start our non-auroc features above this value
binocular	if we should do binocular feature tracking or stereo if there are multiple cameras
histmethod	what type of histogram pre-processing should be done (histogram eq?)
fast_threshold	FAST detection threshold
gridx	size of grid in the x-direction / u-direction
gridy	size of grid in the y-direction / v-direction
minpxdist	features need to be at least this number pixels away from each other
knnratio	matching ratio needed (smaller value forces top two descriptors during match to be more different)

13.46.3 Member Function Documentation

13.46.3.1 feed_monocular()

```
void TrackDescriptor::feed_monocular (
             const CameraData & message,
             size_t msg_id )  [protected]
```

Process a new monocular image.

Parameters

message	Contains our timestamp, images, and camera ids
msg_id	the camera index in message data vector

13.46.3.2 feed_new_camera()

```
void TrackDescriptor::feed_new_camera (
             const CameraData & message )  [virtual]
```

Process a new image.

Parameters

message	Contains our timestamp, images, and camera ids

Implements ov_core::TrackBase.

13.46.3.3 feed_stereo()

```
void TrackDescriptor::feed_stereo (
             const CameraData & message,
             size_t msg_id_left,
             size_t msg_id_right )  [protected]
```

Process new stereo pair of images.

Parameters

message	Contains our timestamp, images, and camera ids
msg_id_left	first image index in message data vector
msg_id_right	second image index in message data vector

13.46.3.4 perform_detection_monocular()

```
void TrackDescriptor::perform_detection_monocular (
             const cv::Mat & img0,
             const cv::Mat & mask0,
             std::vector< cv::KeyPoint > & pts0,
             cv::Mat & desc0,
             std::vector< size_t > & ids0 )  [protected]
```

Detects new features in the current image.

Parameters

img0	image we will detect features on
mask0	mask which has what ROI we do not want features in
pts0	vector of extracted keypoints
desc0	vector of the extracted descriptors
ids0	vector of all new IDs

Given a set of images, and their currently extracted features, this will try to add new features. We return all extracted descriptors here since we DO NOT need to do stereo tracking left to right. Our vector of IDs will be later overwritten when we match features temporally to the previous frame's features. See robust_match() for the matching.

13.46.3.5 perform_detection_stereo()

```
void TrackDescriptor::perform_detection_stereo (
             const cv::Mat & img0,
             const cv::Mat & img1,
             const cv::Mat & mask0,
             const cv::Mat & mask1,
             std::vector< cv::KeyPoint > & pts0,
             std::vector< cv::KeyPoint > & pts1,
             cv::Mat & desc0,
             cv::Mat & desc1,
             size_t cam_id0,
             size_t cam_id1,
             std::vector< size_t > & ids0,
             std::vector< size_t > & ids1 )  [protected]
```

Detects new features in the current stereo pair.

Parameters

img0	left image we will detect features on
img1	right image we will detect features on
mask0	mask which has what ROI we do not want features in
mask1	mask which has what ROI we do not want features in
pts0	left vector of new keypoints
pts1	right vector of new keypoints
desc0	left vector of extracted descriptors
desc1	left vector of extracted descriptors
cam_id0	id of the first camera
cam_id1	id of the second camera
ids0	left vector of all new IDs
ids1	right vector of all new IDs

This does the same logic as the perform_detection_monocular() function, but we also enforce stereo contraints. We also do STEREO matching from the left to right, and only return good matches that are found in both the left and right. Our vector of IDs will be later overwritten when we match features temporally to the previous frame's features. See robust_match() for the matching.

13.46.3.6 robust_match()

```
void TrackDescriptor::robust_match (
            std::vector< cv::KeyPoint > & pts0,
            std::vector< cv::KeyPoint > pts1,
            cv::Mat & desc0,
            cv::Mat & desc1,
            size_t id0,
            size_t id1,
            std::vector< cv::DMatch > & matches )  [protected]
```

Find matches between two keypoint+descriptor sets.

Parameters

pts0	first vector of keypoints
pts1	second vector of keypoints
desc0	first vector of descriptors
desc1	second vector of decriptors
id0	id of the first camera
id1	id of the second camera
matches	vector of matches that we have found

This will perform a "robust match" between the two sets of points (slow but has great results). First we do a simple KNN match from 1to2 and 2to1, which is followed by a ratio check and symmetry check. Original code is from the "Robust↩ Matcher" in the opencv examples, and seems to give very good results in the matches. `https://github.↩`

com/opencv/opencv/blob/master/samples/cpp/tutorial_code/calib3d/real_time_↩
pose_estimation/src/RobustMatcher.cpp

13.47 ov_core::TrackKLT Class Reference

KLT tracking of features.

```
#include <TrackKLT.h>
```

Public Member Functions

- TrackKLT (std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras, int numfeats, int numaruco, bool binocular, HistogramMethod histmethod, int fast_threshold, int gridx, int gridy, int minpxdist)

  Public constructor with configuration variables.
- void feed_new_camera (const CameraData &message)

  Process a new image.

Protected Member Functions

- void feed_monocular (const CameraData &message, size_t msg_id)

  Process a new monocular image.
- void feed_stereo (const CameraData &message, size_t msg_id_left, size_t msg_id_right)

  Process new stereo pair of images.
- void perform_detection_monocular (const std::vector< cv::Mat > &img0pyr, const cv::Mat &mask0, std::vector< cv::KeyPoint > &pts0, std::vector< size_t > &ids0)

  Detects new features in the current image.
- void perform_detection_stereo (const std::vector< cv::Mat > &img0pyr, const std::vector< cv::Mat > &img1pyr, const cv::Mat &mask0, const cv::Mat &mask1, size_t cam_id_left, size_t cam_id_right, std::vector< cv::KeyPoint > &pts0, std::vector< cv::KeyPoint > &pts1, std::vector< size_t > &ids0, std::vector< size_t > &ids1)

  Detects new features in the current stereo pair.
- void perform_matching (const std::vector< cv::Mat > &img0pyr, const std::vector< cv::Mat > &img1pyr, std↩::vector< cv::KeyPoint > &pts0, std::vector< cv::KeyPoint > &pts1, size_t id0, size_t id1, std::vector< uchar > &mask_out)

  KLT track between two images, and do RANSAC afterwards.

Protected Attributes

- int **threshold**
- int **grid_x**
- int **grid_y**
- int **min_px_dist**
- int **pyr_levels** = 3
- cv::Size **win_size** = cv::Size(20, 20)
- std::map< size_t, std::vector< cv::Mat > > **img_pyramid_last**

Additional Inherited Members

13.47.1 Detailed Description

KLT tracking of features.

This is the implementation of a KLT visual frontend for tracking sparse features. We can track either monocular cameras across time (temporally) along with stereo cameras which we also track across time (temporally) but track from left to right to find the stereo correspondence information also. This uses the `calcOpticalFlowPyrLK` OpenCV function to do the KLT tracking.

13.47.2 Constructor & Destructor Documentation

13.47.2.1 TrackKLT()

```
ov_core::TrackKLT::TrackKLT (
            std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras,
            int numfeats,
            int numaruco,
            bool binocular,
            HistogramMethod histmethod,
            int fast_threshold,
            int gridx,
            int gridy,
            int minpxdist )  [inline], [explicit]
```

Public constructor with configuration variables.

Parameters

cameras	camera calibration object which has all camera intrinsics in it
numfeats	number of features we want want to track (i.e. track 200 points from frame to frame)
numaruco	the max id of the arucotags, so we ensure that we start our non-auroc features above this value
binocular	if we should do binocular feature tracking or stereo if there are multiple cameras
histmethod	what type of histogram pre-processing should be done (histogram eq?)
fast_threshold	FAST detection threshold
gridx	size of grid in the x-direction / u-direction
gridy	size of grid in the y-direction / v-direction
minpxdist	features need to be at least this number pixels away from each other

13.47.3 Member Function Documentation

13.47.3.1 feed_monocular()

```
void TrackKLT::feed_monocular (
             const CameraData & message,
             size_t msg_id )  [protected]
```

Process a new monocular image.

Parameters

message	Contains our timestamp, images, and camera ids
msg_id	the camera index in message data vector

13.47.3.2 feed_new_camera()

```
void TrackKLT::feed_new_camera (
             const CameraData & message )  [virtual]
```

Process a new image.

Parameters

message	Contains our timestamp, images, and camera ids

Implements ov_core::TrackBase.

13.47.3.3 feed_stereo()

```
void TrackKLT::feed_stereo (
             const CameraData & message,
             size_t msg_id_left,
             size_t msg_id_right )  [protected]
```

Process new stereo pair of images.

Parameters

message	Contains our timestamp, images, and camera ids
msg_id_left	first image index in message data vector
msg_id_right	second image index in message data vector

13.47.3.4 perform_detection_monocular()

```
void TrackKLT::perform_detection_monocular (
             const std::vector< cv::Mat > & img0pyr,
             const cv::Mat & mask0,
             std::vector< cv::KeyPoint > & pts0,
             std::vector< size_t > & ids0 )  [protected]
```

Detects new features in the current image.

Parameters

img0pyr	image we will detect features on (first level of pyramid)
mask0	mask which has what ROI we do not want features in
pts0	vector of currently extracted keypoints in this image
ids0	vector of feature ids for each currently extracted keypoint

Given an image and its currently extracted features, this will try to add new features if needed. Will try to always have the "max_features" being tracked through KLT at each timestep. Passed images should already be grayscaled.

13.47.3.5 perform_detection_stereo()

```
void TrackKLT::perform_detection_stereo (
             const std::vector< cv::Mat > & img0pyr,
             const std::vector< cv::Mat > & img1pyr,
             const cv::Mat & mask0,
             const cv::Mat & mask1,
             size_t cam_id_left,
             size_t cam_id_right,
             std::vector< cv::KeyPoint > & pts0,
             std::vector< cv::KeyPoint > & pts1,
             std::vector< size_t > & ids0,
             std::vector< size_t > & ids1 )  [protected]
```

Detects new features in the current stereo pair.

Parameters

img0pyr	left image we will detect features on (first level of pyramid)
img1pyr	right image we will detect features on (first level of pyramid)
mask0	mask which has what ROI we do not want features in
mask1	mask which has what ROI we do not want features in
cam_id_left	first camera sensor id
cam_id_right	second camera sensor id
pts0	left vector of currently extracted keypoints
pts1	right vector of currently extracted keypoints
ids0	left vector of feature ids for each currently extracted keypoint
ids1	right vector of feature ids for each currently extracted keypoint

This does the same logic as the perform_detection_monocular() function, but we also enforce stereo contraints. So we detect features in the left image, and then KLT track them onto the right image. If we have valid tracks, then we have both the keypoint on the left and its matching point in the right image. Will try to always have the "max_features" being tracked through KLT at each timestep.

13.47.3.6 perform_matching()

```
void TrackKLT::perform_matching (
            const std::vector< cv::Mat > & img0pyr,
            const std::vector< cv::Mat > & img1pyr,
            std::vector< cv::KeyPoint > & pts0,
            std::vector< cv::KeyPoint > & pts1,
            size_t id0,
            size_t id1,
            std::vector< uchar > & mask_out )  [protected]
```

KLT track between two images, and do RANSAC afterwards.

Parameters

img0pyr	starting image pyramid
img1pyr	image pyramid we want to track too
pts0	starting points
pts1	points we have tracked
id0	id of the first camera
id1	id of the second camera
mask_out	what points had valid tracks

This will track features from the first image into the second image. The two point vectors will be of equal size, but the mask_out variable will specify which points are good or bad. If the second vector is non-empty, it will be used as an initial guess of where the keypoints are in the second image.

13.48 ov_core::TrackSIM Class Reference

Simulated tracker for when we already have uv measurements!

`#include <TrackSIM.h>`

Public Member Functions

- TrackSIM (std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras, int numaruco)

  Public constructor with configuration variables.
- void feed_new_camera (const CameraData &message) override

  Process a new image.
- void feed_measurement_simulation (double timestamp, const std::vector< int > &camids, const std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > &feats)

  Feed function for a synchronized simulated cameras.

Additional Inherited Members

13.48.1 Detailed Description

Simulated tracker for when we already have uv measurements!

This class should be used when we are using the ov_msckf::Simulator class to generate measurements. It simply takes the resulting simulation data and appends it to the internal feature database.

13.48.2 Constructor & Destructor Documentation

13.48.2.1 TrackSIM()

```
ov_core::TrackSIM::TrackSIM (
             std::unordered_map< size_t, std::shared_ptr< CamBase > > cameras,
             int numaruco )  [inline]
```

Public constructor with configuration variables.

Parameters

cameras	camera calibration object which has all camera intrinsics in it
numaruco	the max id of the arucotags, so we ensure that we start our non-auroc features above this value

13.48.3 Member Function Documentation

13.48.3.1 feed_measurement_simulation()

```
void TrackSIM::feed_measurement_simulation (
             double timestamp,
             const std::vector< int > & camids,
             const std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > & feats )
```

Feed function for a synchronized simulated cameras.

Parameters

timestamp	Time that this image was collected
camids	Camera ids that we have simulated measurements for
feats	Raw uv simulated measurements

13.48.3.2 feed_new_camera()

```
void ov_core::TrackSIM::feed_new_camera (
             const CameraData & message )  [inline], [override], [virtual]
```

Process a new image.

Warning

This function should not be used!! Use feed_measurement_simulation() instead.

Parameters

message	Contains our timestamp, images, and camera ids

Implements ov_core::TrackBase.

13.49 ov_type::Type Class Reference

Base class for estimated variables.

```
#include <Type.h>
```

Public Member Functions

- Type (int size_)

  Default constructor for our Type.
- virtual void set_local_id (int new_id)

  Sets id used to track location of variable in the filter covariance.
- int **id** ()

  Access to variable id (i.e. its location in the covariance)
- int **size** ()

  Access to variable size (i.e. its error state size)
- virtual void update (const Eigen::VectorXd &dx)=0

  Update variable due to perturbation of error state.
- virtual const Eigen::MatrixXd & **value** () const

  Access variable's estimate.
- virtual const Eigen::MatrixXd & **fej** () const

  Access variable's first-estimate.
- virtual void set_value (const Eigen::MatrixXd &new_value)

  Overwrite value of state's estimate.

- virtual void set_fej (const Eigen::MatrixXd &new_value)

    Overwrite value of first-estimate.
- virtual std::shared_ptr< Type > clone ()=0

    Create a clone of this variable.
- virtual std::shared_ptr< Type > check_if_subvariable (const std::shared_ptr< Type > check)

    Determine if pass variable is a sub-variable.

Protected Attributes

- Eigen::MatrixXd **_fej**

    First-estimate.
- Eigen::MatrixXd **_value**

    Current best estimate.
- int **_id** = -1

    Location of error state in covariance.
- int **_size** = -1

    Dimension of error state.

13.49.1 Detailed Description

Base class for estimated variables.

This class is used how variables are represented or updated (e.g., vectors or quaternions). Each variable is defined by its error state size and its location in the covariance matrix. We additionally require all sub-types to have a update procedure.

13.49.2 Constructor & Destructor Documentation

13.49.2.1 Type()

```
ov_type::Type::Type (
              int size_ )  [inline]
```

Default constructor for our Type.

Parameters

size← _	degrees of freedom of variable (i.e., the size of the error state)

13.49.3 Member Function Documentation

13.49.3.1 check_if_subvariable()

```
virtual std::shared_ptr< Type > ov_type::Type::check_if_subvariable (
             const std::shared_ptr< Type > check )  [inline], [virtual]
```

Determine if pass variable is a sub-variable.

If the passed variable is a sub-variable or the current variable this will return it. Otherwise it will return a nullptr, meaning that it was unable to be found.

Parameters

check	Type pointer to compare our subvariables to

Reimplemented in ov_type::IMU, and ov_type::PoseJPL.

13.49.3.2 clone()

```
virtual std::shared_ptr< Type > ov_type::Type::clone ( )  [pure virtual]
```

Create a clone of this variable.

Implemented in ov_type::IMU, ov_type::JPLQuat, ov_type::PoseJPL, and ov_type::Vec.

13.49.3.3 set_fej()

```
virtual void ov_type::Type::set_fej (
             const Eigen::MatrixXd & new_value )  [inline], [virtual]
```

Overwrite value of first-estimate.

Parameters

new_value	New value that will overwrite state's fej

Reimplemented in ov_type::IMU, ov_type::JPLQuat, and ov_type::PoseJPL.

13.49.3.4 set_local_id()

```
virtual void ov_type::Type::set_local_id (
             int new_id )  [inline], [virtual]
```

Sets id used to track location of variable in the filter covariance.

Note that the minimum ID is -1 which says that the state is not in our covariance. If the ID is larger than -1 then this is the index location in the covariance matrix.

Parameters

new←_id	entry in filter covariance corresponding to this variable

Reimplemented in ov_type::IMU, and ov_type::PoseJPL.

13.49.3.5 set_value()

```
virtual void ov_type::Type::set_value (
             const Eigen::MatrixXd & new_value )  [inline], [virtual]
```

Overwrite value of state's estimate.

Parameters

new_value	New value that will overwrite state's value

Reimplemented in ov_type::IMU, ov_type::JPLQuat, and ov_type::PoseJPL.

13.49.3.6 update()

```
virtual void ov_type::Type::update (
             const Eigen::VectorXd & dx )  [pure virtual]
```

Update variable due to perturbation of error state.

Parameters

dx	Perturbation used to update the variable through a defined "boxplus" operation

Implemented in ov_type::IMU, ov_type::JPLQuat, ov_type::Landmark, ov_type::PoseJPL, and ov_type::Vec.

13.50 ov_msckf::UpdaterHelper Class Reference

Class that has helper functions for our updaters.

`#include <UpdaterHelper.h>`

Classes

- struct UpdaterHelperFeature

  Feature object that our UpdaterHelper leverages, has all measurements and means.

Static Public Member Functions

- static void get_feature_jacobian_representation (std::shared_ptr< State > state, UpdaterHelperFeature &feature, Eigen::MatrixXd &H_f, std::vector< Eigen::MatrixXd > &H_x, std::vector< std::shared_ptr< ov_type::Type > > &x_order)

  This gets the feature and state Jacobian in respect to the feature representation.
- static void get_feature_jacobian_full (std::shared_ptr< State > state, UpdaterHelperFeature &feature, Eigen::↩ MatrixXd &H_f, Eigen::MatrixXd &H_x, Eigen::VectorXd &res, std::vector< std::shared_ptr< ov_type::Type > > &x_order)

  Will construct the "stacked" Jacobians for a single feature from all its measurements.
- static void nullspace_project_inplace (Eigen::MatrixXd &H_f, Eigen::MatrixXd &H_x, Eigen::VectorXd &res)

  This will project the left nullspace of H_f onto the linear system.
- static void measurement_compress_inplace (Eigen::MatrixXd &H_x, Eigen::VectorXd &res)

  This will perform measurement compression.

13.50.1 Detailed Description

Class that has helper functions for our updaters.

Can compute the Jacobian for a single feature representation. This will create the Jacobian based on what representation our state is in. If we are using the anchor representation then we also have additional Jacobians in respect to the anchor state. Also has functions such as nullspace projection and full jacobian construction. For derivations look at Camera Measurement Update page which has detailed equations.

13.50.2 Member Function Documentation

13.50.2.1 get_feature_jacobian_full()

```
void UpdaterHelper::get_feature_jacobian_full (
            std::shared_ptr< State > state,
            UpdaterHelperFeature & feature,
            Eigen::MatrixXd & H_f,
            Eigen::MatrixXd & H_x,
            Eigen::VectorXd & res,
            std::vector< std::shared_ptr< ov_type::Type > > & x_order )  [static]
```

Will construct the "stacked" Jacobians for a single feature from all its measurements.

Parameters

in	*state*	State of the filter system
in	*feature*	Feature we want to get Jacobians of (must have feature means)
out	*H_f*	Jacobians in respect to the feature error state
out	*H_x*	Extra Jacobians in respect to the state (for example anchored pose)
out	*res*	Measurement residual for this feature
out	*x_order*	Extra variables our extra Jacobian has (for example anchored pose)

13.50.2.2 get_feature_jacobian_representation()

```
void UpdaterHelper::get_feature_jacobian_representation (
            std::shared_ptr< State > state,
            UpdaterHelperFeature & feature,
            Eigen::MatrixXd & H_f,
            std::vector< Eigen::MatrixXd > & H_x,
            std::vector< std::shared_ptr< ov_type::Type > > & x_order )  [static]
```

This gets the feature and state Jacobian in respect to the feature representation.

Parameters

in	*state*	State of the filter system
in	*feature*	Feature we want to get Jacobians of (must have feature means)
out	*H_f*	Jacobians in respect to the feature error state (will be either 3x3 or 3x1 for single depth)
out	*H_x*	Extra Jacobians in respect to the state (for example anchored pose)
out	*x_order*	Extra variables our extra Jacobian has (for example anchored pose)

CASE: Estimate single depth of the feature using the initial bearing

13.50.2.3 measurement_compress_inplace()

```
void UpdaterHelper::measurement_compress_inplace (
            Eigen::MatrixXd & H_x,
            Eigen::VectorXd & res )  [static]
```

This will perform measurement compression.

Please see the Measurement Compression for details on how this works. Note that this is done **in place** so all matrices will be different after a function call.

Parameters

H←→_x	State jacobian
res	Measurement residual

13.50.2.4 nullspace_project_inplace()

```
void UpdaterHelper::nullspace_project_inplace (
            Eigen::MatrixXd & H_f,
            Eigen::MatrixXd & H_x,
            Eigen::VectorXd & res )  [static]
```

This will project the left nullspace of H_f onto the linear system.

Please see the MSCKF Nullspace Projection for details on how this works. This is the MSCKF nullspace projection which removes the dependency on the feature state. Note that this is done **in place** so all matrices will be different after a function call.

Parameters

H←→_f	Jacobian with nullspace we want to project onto the system [res = Hx∗(x-xhat)+Hf(f-fhat)+n]
H←→_x	State jacobian
res	Measurement residual

13.51 ov_msckf::UpdaterHelper::UpdaterHelperFeature Struct Reference

Feature object that our UpdaterHelper leverages, has all measurements and means.

```
#include <UpdaterHelper.h>
```

Public Attributes

- size_t **featid**

  Unique ID of this feature.
- std::unordered_map< size_t, std::vector< Eigen::VectorXf > > **uvs**

  UV coordinates that this feature has been seen from (mapped by camera ID)
- std::unordered_map< size_t, std::vector< Eigen::VectorXf > > **uvs_norm**
- std::unordered_map< size_t, std::vector< double > > **timestamps**

  Timestamps of each UV measurement (mapped by camera ID)

- ov_type::LandmarkRepresentation::Representation **feat_representation**

  What representation our feature is in.
- int **anchor_cam_id** = -1

  What camera ID our pose is anchored in!! By default the first measurement is the anchor.
- double **anchor_clone_timestamp** = -1

  Timestamp of anchor clone.
- Eigen::Vector3d **p_FinA**

  Triangulated position of this feature, in the anchor frame.
- Eigen::Vector3d **p_FinA_fej**

  Triangulated position of this feature, in the anchor frame first estimate.
- Eigen::Vector3d **p_FinG**

  Triangulated position of this feature, in the global frame.
- Eigen::Vector3d **p_FinG_fej**

  Triangulated position of this feature, in the global frame first estimate.

13.51.1 Detailed Description

Feature object that our UpdaterHelper leverages, has all measurements and means.

13.52 ov_msckf::UpdaterMSCKF Class Reference

Will compute the system for our sparse features and update the filter.

```
#include <UpdaterMSCKF.h>
```

Public Member Functions

- UpdaterMSCKF (UpdaterOptions &options, ov_core::FeatureInitializerOptions &feat_init_options)

  Default constructor for our MSCKF updater.
- void update (std::shared_ptr< State > state, std::vector< std::shared_ptr< ov_core::Feature > > &feature_vec)

  Given tracked features, this will try to use them to update the state.

Protected Attributes

- UpdaterOptions **_options**

  Options used during update.
- std::unique_ptr< ov_core::FeatureInitializer > **initializer_feat**

  Feature initializer class object.
- std::map< int, double > **chi_squared_table**

  Chi squared 95th percentile table (lookup would be size of residual)

13.52.1 Detailed Description

Will compute the system for our sparse features and update the filter.

This class is responsible for computing the entire linear system for all features that are going to be used in an update. This follows the original MSCKF, where we first triangulate features, we then nullspace project the feature Jacobian. After this we compress all the measurements to have an efficient update and update the state.

13.52.2 Constructor & Destructor Documentation

13.52.2.1 UpdaterMSCKF()

```
ov_msckf::UpdaterMSCKF::UpdaterMSCKF (
            UpdaterOptions & options,
            ov_core::FeatureInitializerOptions & feat_init_options )  [inline]
```

Default constructor for our MSCKF updater.

Our updater has a feature initializer which we use to initialize features as needed. Also the options allow for one to tune the different parameters for update.

Parameters

options	Updater options (include measurement noise value)
feat_init_options	Feature initializer options

13.52.3 Member Function Documentation

13.52.3.1 update()

```
void UpdaterMSCKF::update (
            std::shared_ptr< State > state,
            std::vector< std::shared_ptr< ov_core::Feature > > & feature_vec )
```

Given tracked features, this will try to use them to update the state.

Parameters

state	State of the filter
feature_vec	Features that can be used for update

Chi2 distance check

13.53 ov_msckf::UpdaterOptions Struct Reference

Struct which stores general updater options.

```
#include <UpdaterOptions.h>
```

Public Member Functions

- void **print** ()

  Nice print function of what parameters we have loaded.

Public Attributes

- double **chi2_multipler** = 5

  What chi-squared multipler we should apply.
- double **sigma_pix** = 1

  Noise sigma for our raw pixel measurements.
- double **sigma_pix_sq** = 1

  Covariance for our raw pixel measurements.

13.53.1 Detailed Description

Struct which stores general updater options.

13.54 ov_msckf::UpdaterSLAM Class Reference

Will compute the system for our sparse SLAM features and update the filter.

```
#include <UpdaterSLAM.h>
```

Public Member Functions

- UpdaterSLAM (UpdaterOptions &options_slam, UpdaterOptions &options_aruco, ov_core::FeatureInitializerOptions &feat_init_options)

  Default constructor for our SLAM updater.
- void update (std::shared_ptr< State > state, std::vector< std::shared_ptr< ov_core::Feature > > &feature_vec)

  Given tracked SLAM features, this will try to use them to update the state.
- void delayed_init (std::shared_ptr< State > state, std::vector< std::shared_ptr< ov_core::Feature > > &feature_vec)

  Given max track features, this will try to use them to initialize them in the state.
- void change_anchors (std::shared_ptr< State > state)

  Will change SLAM feature anchors if it will be marginalized.

Protected Member Functions

- void perform_anchor_change (std::shared_ptr< State > state, std::shared_ptr< ov_type::Landmark > landmark, double new_anchor_timestamp, size_t new_cam_id)

  Shifts landmark anchor to new clone.

Protected Attributes

- UpdaterOptions **_options_slam**

  Options used during update for slam features.
- UpdaterOptions **_options_aruco**

  Options used during update for aruco features.
- std::unique_ptr< ov_core::FeatureInitializer > **initializer_feat**

  Feature initializer class object.
- std::map< int, double > **chi_squared_table**

  Chi squared 95th percentile table (lookup would be size of residual)

13.54.1 Detailed Description

Will compute the system for our sparse SLAM features and update the filter.

This class is responsible for performing delayed feature initialization, SLAM update, and SLAM anchor change for anchored feature representations.

13.54.2 Constructor & Destructor Documentation

13.54.2.1 UpdaterSLAM()

```
ov_msckf::UpdaterSLAM::UpdaterSLAM (
            UpdaterOptions & options_slam,
            UpdaterOptions & options_aruco,
            ov_core::FeatureInitializerOptions & feat_init_options )  [inline]
```

Default constructor for our SLAM updater.

Our updater has a feature initializer which we use to initialize features as needed. Also the options allow for one to tune the different parameters for update.

Parameters

options_slam	Updater options (include measurement noise value) for SLAM features
options_aruco	Updater options (include measurement noise value) for ARUCO features
feat_init_options	Feature initializer options

13.54.3 Member Function Documentation

13.54.3.1 change_anchors()

```
void UpdaterSLAM::change_anchors (
            std::shared_ptr< State > state )
```

Will change SLAM feature anchors if it will be marginalized.

Makes sure that if any clone is about to be marginalized, it changes anchor representation. By default, this will shift the anchor into the newest IMU clone and keep the camera calibration anchor the same.

Parameters

state	State of the filter

13.54.3.2 delayed_init()

```
void UpdaterSLAM::delayed_init (
            std::shared_ptr< State > state,
            std::vector< std::shared_ptr< ov_core::Feature > > & feature_vec )
```

Given max track features, this will try to use them to initialize them in the state.

Parameters

state	State of the filter
feature_vec	Features that can be used for update

13.54.3.3 perform_anchor_change()

```
void UpdaterSLAM::perform_anchor_change (
            std::shared_ptr< State > state,
            std::shared_ptr< ov_type::Landmark > landmark,
            double new_anchor_timestamp,
            size_t new_cam_id )  [protected]
```

Shifts landmark anchor to new clone.

Parameters

state	State of filter
landmark	landmark whose anchor is being shifter
new_anchor_timestamp	Clone timestamp we want to move to
new_cam_id	Which camera frame we want to move to

13.54.3.4 update()

```
void UpdaterSLAM::update (
             std::shared_ptr< State > state,
             std::vector< std::shared_ptr< ov_core::Feature > > & feature_vec )
```

Given tracked SLAM features, this will try to use them to update the state.

Parameters

state	State of the filter
feature_vec	Features that can be used for update

13.55 ov_msckf::UpdaterZeroVelocity Class Reference

Will try to *detect* and then update using zero velocity assumption.

```
#include <UpdaterZeroVelocity.h>
```

Public Member Functions

- UpdaterZeroVelocity (UpdaterOptions &options, Propagator::NoiseManager &noises, std::shared_ptr< ov_core::FeatureDatabase > db, std::shared_ptr< Propagator > prop, double gravity_mag, double zupt_↩ max_velocity, double zupt_noise_multiplier, double zupt_max_disparity)

  Default constructor for our zero velocity detector and updater.
- void feed_imu (const ov_core::ImuData &message)

  Feed function for inertial data.
- bool try_update (std::shared_ptr< State > state, double timestamp)

  Will first detect if the system is zero velocity, then will update.

Protected Attributes

- UpdaterOptions **_options**

  Options used during update (chi2 multiplier)
- Propagator::NoiseManager **_noises**

  Container for the imu noise values.
- std::shared_ptr< ov_core::FeatureDatabase > **_db**

  Feature tracker database with all features in it.
- std::shared_ptr< Propagator > **_prop**

  Our propagator!
- Eigen::Vector3d **_gravity**

  Gravity vector.
- double **_zupt_max_velocity** = 1.0

  Max velocity (m/s) that we should consider a zupt with.
- double **_zupt_noise_multiplier** = 1.0

  Multiplier of our IMU noise matrix (default should be 1.0)
- double **_zupt_max_disparity** = 1.0

  Max disparity (pixels) that we should consider a zupt with.
- std::map< int, double > **chi_squared_table**

  Chi squared 95th percentile table (lookup would be size of residual)
- std::vector< ov_core::ImuData > **imu_data**

  Our history of IMU messages (time, angular, linear)
- double **last_prop_time_offset** = 0.0

  Estimate for time offset at last propagation time.
- bool **have_last_prop_time_offset** = false
- double **last_zupt_state_timestamp** = 0.0

  Last timestamp we did zero velocity update with.

13.55.1 Detailed Description

Will try to *detect* and then update using zero velocity assumption.

Consider the case that a VIO unit remains stationary for a period time. Typically this can cause issues in a monocular system without SLAM features since no features can be triangulated. Additional, if features could be triangulated (e.g. stereo) the quality can be poor and hurt performance. If we can detect the cases where we are stationary then we can leverage this to prevent the need to do feature update during this period. The main application would be using this on a **wheeled vehicle** which needs to stop (e.g. stop lights or parking).

13.55.2 Constructor & Destructor Documentation

13.55.2.1 UpdaterZeroVelocity()

```
ov_msckf::UpdaterZeroVelocity::UpdaterZeroVelocity (
            UpdaterOptions & options,
            Propagator::NoiseManager & noises,
            std::shared_ptr< ov_core::FeatureDatabase > db,
            std::shared_ptr< Propagator > prop,
            double gravity_mag,
            double zupt_max_velocity,
            double zupt_noise_multiplier,
            double zupt_max_disparity )  [inline]
```

Default constructor for our zero velocity detector and updater.

Parameters

options	Updater options (chi2 multiplier)
noises	imu noise characteristics (continuous time)
db	Feature tracker database with all features in it
prop	Propagator class object which can predict the state forward in time
gravity_mag	Global gravity magnitude of the system (normally 9.81)
zupt_max_velocity	Max velocity we should consider to do a update with
zupt_noise_multiplier	Multiplier of our IMU noise matrix (default should be 1.0)
zupt_max_disparity	Max disparity we should consider to do a update with

13.55.3 Member Function Documentation

13.55.3.1 feed_imu()

```
void ov_msckf::UpdaterZeroVelocity::feed_imu (
            const ov_core::ImuData & message )  [inline]
```

Feed function for inertial data.

Parameters

message	Contains our timestamp and inertial information

13.55.3.2 try_update()

```
bool UpdaterZeroVelocity::try_update (
                std::shared_ptr< State > state,
                double timestamp )
```

Will first detect if the system is zero velocity, then will update.

Parameters

state	State of the filter
timestamp	Next camera timestamp we want to see if we should propagate to.

Returns

True if the system is currently at zero velocity

13.56 ov_type::Vec Class Reference

Derived Type class that implements vector variables.

```
#include <Vec.h>
```

Public Member Functions

- Vec (int dim)

  Default constructor for Vec.
- void update (const Eigen::VectorXd &dx) override

  Implements the update operation through standard vector addition.
- std::shared_ptr< Type > clone () override

  Performs all the cloning.

Additional Inherited Members

13.56.1 Detailed Description

Derived Type class that implements vector variables.

13.56.2 Constructor & Destructor Documentation

13.56.2.1 Vec()

```
ov_type::Vec::Vec (
                int dim )  [inline]
```

Default constructor for Vec.

Parameters

dim	Size of the vector (will be same as error state)

13.56.3 Member Function Documentation

13.56.3.1 clone()

```
std::shared_ptr< Type > ov_type::Vec::clone ( )  [inline], [override], [virtual]
```

Performs all the cloning.

Implements ov_type::Type.

13.56.3.2 update()

```
void ov_type::Vec::update (
             const Eigen::VectorXd & dx )  [inline], [override], [virtual]
```

Implements the update operation through standard vector addition.

Parameters

dx	Additive error state correction

Implements ov_type::Type.

13.57 ov_msckf::VioManager Class Reference

Core class that manages the entire system.

```
#include <VioManager.h>
```

Public Member Functions

- VioManager (VioManagerOptions &params_)
  Default constructor, will load all configuration variables.

- void feed_measurement_imu (const ov_core::ImuData &message)

    Feed function for inertial data.
- void feed_measurement_camera (const ov_core::CameraData &message)

    Feed function for camera measurements.
- void feed_measurement_simulation (double timestamp, const std::vector< int > &camids, const std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > &feats)

    Feed function for a synchronized simulated cameras.
- void initialize_with_gt (Eigen::Matrix< double, 17, 1 > imustate)

    Given a state, this will initialize our IMU state.
- bool **initialized** ()

    If we are initialized or not.
- double **initialized_time** ()

    Timestamp that the system was initialized at.
- VioManagerOptions **get_params** ()

    Accessor for current system parameters.
- std::shared_ptr< State > **get_state** ()

    Accessor to get the current state.
- std::shared_ptr< Propagator > **get_propagator** ()

    Accessor to get the current propagator.
- cv::Mat **get_historical_viz_image** ()

    Get a nice visualization image of what tracks we have.
- std::vector< Eigen::Vector3d > **get_good_features_MSCKF** ()

    Returns 3d features used in the last update in global frame.
- std::vector< Eigen::Vector3d > **get_features_SLAM** ()

    Returns 3d SLAM features in the global frame.
- std::vector< Eigen::Vector3d > **get_features_ARUCO** ()

    Returns 3d ARUCO features in the global frame.
- void **get_active_image** (double &timestamp, cv::Mat &image)

    Return the image used when projecting the active tracks.
- void **get_active_tracks** (double &timestamp, std::unordered_map< size_t, Eigen::Vector3d > &feat_posinG, std::unordered_map< size_t, Eigen::Vector3d > &feat_tracks_uvd)

    Returns active tracked features in the current frame.

Protected Member Functions

- void track_image_and_update (const ov_core::CameraData &message)

    Given a new set of camera images, this will track them.
- void do_feature_propagate_update (const ov_core::CameraData &message)

    This will do the propagation and feature updates to the state.
- bool try_to_initialize ()

    This function will try to initialize the state.
- void retriangulate_active_tracks (const ov_core::CameraData &message)

    This function will will re-triangulate all features in the current frame.

Protected Attributes

- VioManagerOptions **params**

  Manager parameters.
- std::shared_ptr< State > **state**

  Our master state object :D.
- std::shared_ptr< Propagator > **propagator**

  Propagator of our state.
- std::shared_ptr< ov_core::FeatureDatabase > **trackDATABASE**

  Complete history of our feature tracks.
- std::shared_ptr< ov_core::TrackBase > **trackFEATS**

  Our sparse feature tracker (klt or descriptor)
- std::shared_ptr< ov_core::TrackBase > **trackARUCO**

  Our aruoc tracker.
- std::shared_ptr< ov_init::InertialInitializer > **initializer**

  State initializer.
- bool **is_initialized_vio** = false

  Boolean if we are initialized or not.
- std::shared_ptr< UpdaterMSCKF > **updaterMSCKF**

  Our MSCKF feature updater.
- std::shared_ptr< UpdaterSLAM > **updaterSLAM**

  Our SLAM/ARUCO feature updater.
- std::shared_ptr< UpdaterZeroVelocity > **updaterZUPT**

  Our zero velocity tracker.
- std::deque< ov_core::CameraData > camera_queue
- std::ofstream **of_statistics**
- boost::posix_time::ptime **rT1**
- boost::posix_time::ptime **rT2**
- boost::posix_time::ptime **rT3**
- boost::posix_time::ptime **rT4**
- boost::posix_time::ptime **rT5**
- boost::posix_time::ptime **rT6**
- boost::posix_time::ptime **rT7**
- double **timelastupdate** = -1
- double **distance** = 0
- double **startup_time** = -1
- bool **did_zupt_update** = false
- cv::Mat **zupt_image**
- std::map< size_t, cv::Mat > **zupt_img_last**
- bool **has_moved_since_zupt** = false
- std::vector< Eigen::Vector3d > **good_features_MSCKF**
- std::shared_ptr< ov_core::FeatureInitializer > **active_tracks_initializer**

  Feature initializer used to triangulate all active tracks.
- double **active_tracks_time** = -1
- std::unordered_map< size_t, Eigen::Vector3d > **active_tracks_posinG**
- std::unordered_map< size_t, Eigen::Vector3d > **active_tracks_uvd**
- cv::Mat **active_image**

13.57.1 Detailed Description

Core class that manages the entire system.

This class contains the state and other algorithms needed for the MSCKF to work. We feed in measurements into this class and send them to their respective algorithms. If we have measurements to propagate or update with, this class will call on our state to do that.

13.57.2 Constructor & Destructor Documentation

13.57.2.1 VioManager()

```
VioManager::VioManager (
            VioManagerOptions & params_ )
```

Default constructor, will load all configuration variables.

Parameters

params↩_	Parameters loaded from either ROS or CMDLINE

13.57.3 Member Function Documentation

13.57.3.1 do_feature_propagate_update()

```
void VioManager::do_feature_propagate_update (
            const ov_core::CameraData & message )  [protected]
```

This will do the propagation and feature updates to the state.

Parameters

message	Contains our timestamp, images, and camera ids

13.57.3.2 feed_measurement_camera()

```
void ov_msckf::VioManager::feed_measurement_camera (
             const ov_core::CameraData & message )  [inline]
```

Feed function for camera measurements.

Parameters

message	Contains our timestamp, images, and camera ids

13.57.3.3 feed_measurement_imu()

```
void VioManager::feed_measurement_imu (
             const ov_core::ImuData & message )
```

Feed function for inertial data.

Parameters

message	Contains our timestamp and inertial information

13.57.3.4 feed_measurement_simulation()

```
void VioManager::feed_measurement_simulation (
             double timestamp,
             const std::vector< int > & camids,
             const std::vector< std::vector< std::pair< size_t, Eigen::VectorXf > > > & feats )
```

Feed function for a synchronized simulated cameras.

Parameters

timestamp	Time that this image was collected
camids	Camera ids that we have simulated measurements for
feats	Raw uv simulated measurements

13.57.3.5 initialize_with_gt()

```
void ov_msckf::VioManager::initialize_with_gt (
            Eigen::Matrix< double, 17, 1 > imustate )  [inline]
```

Given a state, this will initialize our IMU state.

Parameters

imustate	State in the MSCKF ordering: [time(sec),q_GtoI,p_IinG,v_IinG,b_gyro,b_accel]

13.57.3.6 retriangulate_active_tracks()

```
void VioManager::retriangulate_active_tracks (
            const ov_core::CameraData & message )  [protected]
```

This function will will re-triangulate all features in the current frame.

For all features that are currently being tracked by the system, this will re-triangulate them. This is useful for downstream applications which need the current pointcloud of points (e.g. loop closure). This will try to triangulate *all* points, not just ones that have been used in the update.

Parameters

message	Contains our timestamp, images, and camera ids

13.57.3.7 track_image_and_update()

```
void VioManager::track_image_and_update (
            const ov_core::CameraData & message )  [protected]
```

Given a new set of camera images, this will track them.

If we are having stereo tracking, we should call stereo tracking functions. Otherwise we will try to track on each of the images passed.

Parameters

message	Contains our timestamp, images, and camera ids

13.57.3.8 try_to_initialize()

```
bool VioManager::try_to_initialize ( )  [protected]
```

This function will try to initialize the state.

This should call on our initializer and try to init the state. In the future we should call the structure-from-motion code from here. This function could also be repurposed to re-initialize the system after failure.

Returns

True if we have successfully initialized

13.57.4 Member Data Documentation

13.57.4.1 camera_queue

```
std::deque<ov_core::CameraData> ov_msckf::VioManager::camera_queue  [protected]
```

Queue up camera measurements sorted by time and trigger once we have exactly one IMU measurement with timestamp newer than the camera measurement This also handles out-of-order camera measurements, which is rare, but a nice feature to have for general robustness to bad camera drivers.

13.58 ov_msckf::VioManagerOptions Struct Reference

Struct which stores all options needed for state estimation.

```
#include <VioManagerOptions.h>
```

Public Member Functions

- void print_and_load (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load the non-simulation parameters of the system and print.
- void print_and_load_estimator (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load print out all estimator settings loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.
- void print_and_load_noise (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load print out all noise parameters loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.
- void print_and_load_state (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load and print all state parameters (e.g. sensor extrinsics) This allows for visual checking that everything was loaded properly from ROS/CMD parsers.
- void print_and_load_trackers (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load print out all parameters related to visual tracking This allows for visual checking that everything was loaded properly from ROS/CMD parsers.
- void print_and_load_simulation (const std::shared_ptr< ov_core::YamlParser > &parser=nullptr)

  This function will load print out all simulated parameters. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Public Attributes

- StateOptions **state_options**

    Core state options (e.g. number of cameras, use fej, stereo, what calibration to enable etc)
- ov_init::InertialInitializerOptions **init_options**

    Our state initialization options (e.g. window size, num features, if we should get the calibration)
- double **dt_slam_delay** = 2.0

    Delay, in seconds, that we should wait from init before we start estimating SLAM features.
- bool **try_zupt** = false

    If we should try to use zero velocity update.
- double **zupt_max_velocity** = 1.0

    Max velocity we will consider to try to do a zupt (i.e. if above this, don't do zupt)
- double **zupt_noise_multiplier** = 1.0

    Multiplier of our zupt measurement IMU noise matrix (default should be 1.0)
- double **zupt_max_disparity** = 1.0

    Max disparity we will consider to try to do a zupt (i.e. if above this, don't do zupt)
- bool **zupt_only_at_beginning** = false

    If we should only use the zupt at the very beginning static initialization phase.
- bool **record_timing_information** = false

    If we should record the timing performance to file.
- std::string **record_timing_filepath** = "ov_msckf_timing.txt"

    The path to the file we will record the timing information into.
- Propagator::NoiseManager **imu_noises**

    IMU noise (gyroscope and accelerometer)
- UpdaterOptions **msckf_options**

    Update options for MSCKF features (pixel noise and chi2 multiplier)
- UpdaterOptions **slam_options**

    Update options for SLAM features (pixel noise and chi2 multiplier)
- UpdaterOptions **aruco_options**

    Update options for ARUCO features (pixel noise and chi2 multiplier)
- UpdaterOptions **zupt_options**

    Update options for zero velocity (chi2 multiplier)
- double **gravity_mag** = 9.81

    Gravity magnitude in the global frame (i.e. should be 9.81 typically)
- double **calib_camimu_dt** = 0.0

    Time offset between camera and IMU.
- std::unordered_map< size_t, std::shared_ptr< ov_core::CamBase > > **camera_intrinsics**

    Map between camid and camera intrinsics (fx, fy, cx, cy, d1...d4, cam_w, cam_h)
- std::map< size_t, Eigen::VectorXd > **camera_extrinsics**

    Map between camid and camera extrinsics (q_ItoC, p_IinC).
- bool **use_mask** = false

    If we should try to load a mask and use it to reject invalid features.
- std::map< size_t, cv::Mat > **masks**

    Mask images for each camera.
- bool **use_stereo** = true

    If we should process two cameras are being stereo or binocular. If binocular, we do monocular feature tracking on each image.

- bool **use_klt** = true

  If we should use KLT tracking, or descriptor matcher.
- bool **use_aruco** = true

  If should extract aruco tags and estimate them.
- bool **downsize_aruco** = true

  Will half the resolution of the aruco tag image (will be faster)
- bool **downsample_cameras** = false

  Will half the resolution all tracking image (aruco will be 1/4 instead of halved if dowsize_aruoc also enabled)
- bool **use_multi_threading** = true

  If our front-end should try to use some multi-threading for stereo matching.
- int **num_pts** = 150

  The number of points we should extract and track in each image frame. This highly effects the computation required for tracking.
- int **fast_threshold** = 20

  Fast extraction threshold.
- int **grid_x** = 5

  Number of grids we should split column-wise to do feature extraction in.
- int **grid_y** = 5

  Number of grids we should split row-wise to do feature extraction in.
- int **min_px_dist** = 10

  Will check after doing KLT track and remove any features closer than this.
- ov_core::TrackBase::HistogramMethod **histogram_method** = ov_core::TrackBase::HistogramMethod::↩ HISTOGRAM

  What type of pre-processing histogram method should be applied to images.
- double **knn_ratio** = 0.85

  KNN ration between top two descriptor matcher which is required to be a good match.
- ov_core::FeatureInitializerOptions **featinit_options**

  Parameters used by our feature initialize / triangulator.
- int **sim_seed_state_init** = 0

  Seed for initial states (i.e. random feature 3d positions in the generated map)
- int **sim_seed_preturb** = 0

  Seed for calibration perturbations. Change this to perturb by different random values if perturbations are enabled.
- int sim_seed_measurements = 0
- bool **sim_do_perturbation** = false

  If we should perturb the calibration that the estimator starts with.
- std::string **sim_traj_path** = "src/open_vins/ov_data/sim/udel_gore.txt"

  Path to the trajectory we will b-spline and simulate on. Should be time(s),pos(xyz),ori(xyzw) format.
- double sim_distance_threshold = 1.2
- double **sim_freq_cam** = 10.0

  Frequency (Hz) that we will simulate our cameras.
- double **sim_freq_imu** = 400.0

  Frequency (Hz) that we will simulate our inertial measurement unit.
- double **sim_min_feature_gen_distance** = 5

  Feature distance we generate features from (minimum)
- double **sim_max_feature_gen_distance** = 10

  Feature distance we generate features from (maximum)

13.58.1 Detailed Description

Struct which stores all options needed for state estimation.

This is broken into a few different parts: estimator, trackers, and simulation. If you are going to add a parameter here you will need to add it to the parsers. You will also need to add it to the print statement at the bottom of each.

13.58.2 Member Function Documentation

13.58.2.1 print_and_load()

```
void ov_msckf::VioManagerOptions::print_and_load (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load the non-simulation parameters of the system and print.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.2.2 print_and_load_estimator()

```
void ov_msckf::VioManagerOptions::print_and_load_estimator (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load print out all estimator settings loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.2.3 print_and_load_noise()

```
void ov_msckf::VioManagerOptions::print_and_load_noise (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load print out all noise parameters loaded. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.2.4 print_and_load_simulation()

```
void ov_msckf::VioManagerOptions::print_and_load_simulation (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load print out all simulated parameters. This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.2.5 print_and_load_state()

```
void ov_msckf::VioManagerOptions::print_and_load_state (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load and print all state parameters (e.g. sensor extrinsics) This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.2.6 print_and_load_trackers()

```
void ov_msckf::VioManagerOptions::print_and_load_trackers (
            const std::shared_ptr< ov_core::YamlParser > & parser = nullptr )  [inline]
```

This function will load print out all parameters related to visual tracking This allows for visual checking that everything was loaded properly from ROS/CMD parsers.

Parameters

parser	If not null, this parser will be used to load our parameters

13.58.3 Member Data Documentation

13.58.3.1 sim_distance_threshold

```
double ov_msckf::VioManagerOptions::sim_distance_threshold = 1.2
```

We will start simulating after we have moved this much along the b-spline. This prevents static starts as we init from groundtruth in simulation.

13.58.3.2 sim_seed_measurements

```
int ov_msckf::VioManagerOptions::sim_seed_measurements = 0
```

Measurement noise seed. This should be incremented for each run in the Monte-Carlo simulation to generate the same true measurements, but diffferent noise values.

13.59 ov_core::YamlParser Class Reference

Helper class to do OpenCV yaml parsing from both file and ROS.

```
#include <opencv_yaml_parse.h>
```

Public Member Functions

- YamlParser (const std::string &config_path, bool fail_if_not_found=true)
  Constructor that loads all three configuration files.
- std::string get_config_folder ()
  Will get the folder this config file is in.
- bool successful () const
  Check to see if all parameters were read succesfully.
- template<class T >
  void parse_config (const std::string &node_name, T &node_result, bool required=true)
  Custom parser for the ESTIMATOR parameters.
- template<class T >
  void parse_external (const std::string &external_node_name, const std::string &sensor_name, const std::string &node_name, T &node_result, bool required=true)
  Custom parser for the external parameter files with levels.
- void parse_external (const std::string &external_node_name, const std::string &sensor_name, const std::string &node_name, Eigen::Matrix3d &node_result, bool required=true)
  Custom parser for Matrix3d in the external parameter files with levels.
- void parse_external (const std::string &external_node_name, const std::string &sensor_name, const std::string &node_name, Eigen::Matrix4d &node_result, bool required=true)
  Custom parser for Matrix4d in the external parameter files with levels.

13.59.1 Detailed Description

Helper class to do OpenCV yaml parsing from both file and ROS.

The logic is as follows:

- Given a path to the main config file we will load it into our cv::FileStorage object.
- From there the user can request for different parameters of different types from the config.
- If we have ROS, then we will also check to see if the user has overridden any config files via ROS.
- The ROS parameters always take priority over the ones in our config.

NOTE: There are no "nested" yaml parameters. They are all under the "root" of the yaml file!!! NOTE: The camera and imu have nested, but those are handled externally....

13.59.2 Constructor & Destructor Documentation

13.59.2.1 YamlParser()

```
ov_core::YamlParser::YamlParser (
            const std::string & config_path,
            bool fail_if_not_found = true )  [inline], [explicit]
```

Constructor that loads all three configuration files.

Parameters

config_path	Path to the YAML file we will parse
fail_if_not_found	If we should terminate the program if we can't open the config file

13.59.3 Member Function Documentation

13.59.3.1 get_config_folder()

```
std::string ov_core::YamlParser::get_config_folder ( )  [inline]
```

Will get the folder this config file is in.

Returns

Config folder

13.59.3.2 parse_config()

```
template<class T >
void ov_core::YamlParser::parse_config (
            const std::string & node_name,
            T & node_result,
            bool required = true )  [inline]
```

Custom parser for the ESTIMATOR parameters.

This will load the data from the main config file. If it is unable it will give a warning to the user it couldn't be found.

Template Parameters

T	Type of parameter we are looking for.

Parameters

node_name	Name of the node
node_result	Resulting value (should already have default value in it)
required	If this parameter is required by the user to set

13.59.3.3 parse_external() [1/3]

```
void ov_core::YamlParser::parse_external (
            const std::string & external_node_name,
            const std::string & sensor_name,
            const std::string & node_name,
            Eigen::Matrix3d & node_result,
            bool required = true )  [inline]
```

Custom parser for Matrix3d in the external parameter files with levels.

This will first load the external file requested. From there it will try to find the first level requested (e.g. imu0, cam0, cam1). Then the requested node can be found under this sensor name. ROS can override the config with `<sensor←_name>_<node_name>` (e.g., cam0_distortion).

Parameters

external_node_name	Name of the node we will get our relative path from
sensor_name	The first level node we will try to get the requested node under
node_name	Name of the node
node_result	Resulting value (should already have default value in it)
required	If this parameter is required by the user to set

13.59.3.4 parse_external() [2/3]

```
void ov_core::YamlParser::parse_external (
            const std::string & external_node_name,
            const std::string & sensor_name,
            const std::string & node_name,
            Eigen::Matrix4d & node_result,
            bool required = true )  [inline]
```

Custom parser for Matrix4d in the external parameter files with levels.

This will first load the external file requested. From there it will try to find the first level requested (e.g. imu0, cam0, cam1). Then the requested node can be found under this sensor name. ROS can override the config with `<sensor_name>_<node_name>` (e.g., cam0_distortion).

Parameters

external_node_name	Name of the node we will get our relative path from
sensor_name	The first level node we will try to get the requested node under
node_name	Name of the node
node_result	Resulting value (should already have default value in it)
required	If this parameter is required by the user to set

13.59.3.5 parse_external() [3/3]

```
template<class T >
void ov_core::YamlParser::parse_external (
            const std::string & external_node_name,
            const std::string & sensor_name,
            const std::string & node_name,
            T & node_result,
            bool required = true )  [inline]
```

Custom parser for the external parameter files with levels.

This will first load the external file requested. From there it will try to find the first level requested (e.g. imu0, cam0, cam1). Then the requested node can be found under this sensor name. ROS can override the config with `<sensor↩ _name>_<node_name>` (e.g., cam0_distortion).

Template Parameters

T	Type of parameter we are looking for.

Parameters

external_node_name	Name of the node we will get our relative path from
sensor_name	The first level node we will try to get the requested node under
node_name	Name of the node
node_result	Resulting value (should already have default value in it)
required	If this parameter is required by the user to set

13.59.3.6 successful()

```
bool ov_core::YamlParser::successful ( ) const  [inline]
```

Check to see if all parameters were read succesfully.

Returns

True if we found all parameters

Bibliography

Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 35(10): 1157–1163, 2016. 11, 13

Pavel Davidson, Jani Hautamäki, Jussi Collin, and Jarmo Takala. Improved vehicle positioning in urban environment through integration of gps and low-cost inertial sensors. In *Proceedings of the European Navigation Conference (ENC), Naples, Italy*, pages 3–6, 2009. URL `http://www.tkt.cs.tut.fi/research/nappo_files/1_C2.pdf`. 52

Jeffrey Delmerico and Davide Scaramuzza. A benchmark comparison of monocular visual-inertial odometry algorithms for flying robots. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2502–2509. IEEE, 2018. URL `http://rpg.ifi.uzh.ch/docs/ICRA18_Delmerico.pdf`. 94

Tue-Cuong Dong-Si and Anastasios I Mourikis. Estimator initialization in vision-aided inertial navigation with unknown camera-imu calibration. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1064–1071. IEEE, 2012. 154

Kevin Eckenhoff, Patrick Geneva, and Guoquan Huang. Closed-form preintegration methods for graph-based visual-inertial navigation. *International Journal of Robotics Research*, 38(5), 2019. doi: 10.1177/0278364919835021. URL `https://doi.org/10.1177/0278364919835021`. 72, 83, 127, 130, 132

Paul Furgale, Joern Rehder, and Roland Siegwart. Unified temporal and spatial calibration for multi-sensor systems. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1280–1286. IEEE, 2013. 17

Guoquan P Huang, Anastasios I Mourikis, and Stergios I Roumeliotis. Observability-based rules for designing consistent ekf slam estimators. *The International Journal of Robotics Research*, 29(5):502–528, 2010. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.228.5851&rep=rep1&type=pdf`. 29

Jinwoo Jeon, Sungwook Jung, Eungchang Lee, Duckyu Choi, and Hyun Myung. Run your visual-inertial odometry on nvidia jetson: Benchmark tests on a micro aerial vehicle. *IEEE Robotics and Automation Letters*, 6(3):5332–5339, 2021. 16

Jinyong Jeong, Younggun Cho, Young-Sik Shin, Hyunchul Roh, and Ayoung Kim. Complex urban dataset with multi-level sensors from highly diverse urban environments. *The International Journal of Robotics Research*, 38(6):642–657, 2019. 16

Steven M Kay. *Fundamentals of statistical signal processing*. Prentice Hall PTR, 1993. URL `http://users.isr.ist.utl.pt/~pjcro/temp/Fundamentals%20Of%20Statistical%20Signal%20Processing%2D%2DEstimation%20Theory-Kay.pdf`. 31

Mingyang Li. *Visual-inertial odometry on resource-constrained systems*. PhD thesis, UC Riverside, 2014. URL `https://escholarship.org/uc/item/4nn0j264`. 47

Mingyang Li and Anastasios I Mourikis. High-precision, consistent ekf-based visual-inertial odometry. *The International Journal of Robotics Research*, 32(6):690–711, 2013. 29

Peter S Maybeck. *Stochastic models, estimation, and control*, volume 3. Academic press, 1982. URL https://books.google.com/books?id=L_YVMUJKNQUC. 23

Anastasios I Mourikis and Stergios I Roumeliotis. A multi-state constraint kalman filter for vision-aided inertial navigation. In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 3565–3572. IEEE, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.437.1085&rep=rep1&type=pdf. 46, 47, 96

E. Mueggler, G. Gallego, H. Rebecq, and D. Scaramuzza. Continuous-time visual-inertial odometry for event cameras. *IEEE Transactions on Robotics*, pages 1–16, 2018. URL http://rpg.ifi.uzh.ch/docs/TRO18_Mueggler.pdf. 55

Alonso Patron-Perez, Steven Lovegrove, and Gabe Sibley. A spline-based trajectory representation for sensor fusion and rolling shutter cameras. *International Journal of Computer Vision*, 113(3):208–219, 2015. 55

Tong Qin, Peiliang Li, and Shaojie Shen. Vins-mono: A robust and versatile monocular visual-inertial state estimator. *IEEE Transactions on Robotics*, 34(4):1004–1020, 2018. URL https://arxiv.org/pdf/1708.03852.pdf. 47

Arvind Ramanandan, Anning Chen, and Jay A Farrell. Inertial navigation aiding by stationary updates. *IEEE Transactions on Intelligent Transportation Systems*, 13(1):235–248, 2011. 52

David Schubert, Thore Goll, Nikolaus Demmel, Vladyslav Usenko, J ö rg St ü ckler, and Daniel Cremers. The tum vi benchmark for evaluating visual-inertial odometry. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1680–1687. IEEE, 2018. URL https://arxiv.org/pdf/1804.06120.pdf. 14, 15

Nikolas Trawny and Stergios I Roumeliotis. Indirect kalman filter for 3d attitude estimation. *University of Minnesota, Dept. of Comp. Sci. & Eng., Tech. Rep*, 2:2005, 2005. URL http://mars.cs.umn.edu/tr/reports/Trawny05b.pdf. 22, 25, 27, 42

Brandon Wagstaff, Valentin Peretroukhin, and Jonathan Kelly. Improving foot-mounted inertial navigation through real-time motion classification. In *2017 International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, pages 1–8. IEEE, 2017. URL https://arxiv.org/pdf/1707.01152.pdf. 52

Zichao Zhang and Davide Scaramuzza. A tutorial on quantitative trajectory evaluation for visual(-inertial) odometry. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 7244–7251. IEEE, 2018. URL http://rpg.ifi.uzh.ch/docs/IROS18_Zhang.pdf. 58, 94

Index