# Real-time State Estimation Using Monocular Camera and IMU Based on Vins-Mono


**Daniel Roth, MSc student**

**Under supervision of Yonatan Mendel**

Department of Electrical Engineering, Tel-Aviv University

# Table of content

# Abstract

This project book is an experiment on state estimation algorithm based on visual and inertial data using a miniature drone.

The purpose of the project is achieve a reliable drone state estimation using only the onboard IMU and an add-on camera. The algorithm used is VINS-MONO [1] and the selected drone is Crazyflie that was mounted with a small low-cost RGB camera.

Drone pose estimation is a task that was solved in numerous ways – the contribution of this project is the use of the VINS-MONO [1] algorithm along with the low cost setup of the Crazyflie and the camera. To our knowledge this was not done before.

This project book will elaborate on the theoretical and software stages of the algorithm, the software environments used, the hardware that was used and failures and solutions that we encountered through the project, to finish, we will present the benefits of the setup and our experiments and will discuss possible future steps.

# Introduction

In the field on robotics it is crucial to be able to place the robot in some coordinate system, and preferably in a Cartesian one for a more intuitive use.
After the robot is placed in such coordinate system, the operator can navigate toward the designated location.

There are different approaches to locate the position of the robot – in some, there is prior knowledge on the surrounding, and some uses GPS signals.

When there is no prior and GPS can't be relied on – most approaches uses a mounted camera.
With a camera the robot can scan its surrounding and allocate key feature points. With these points a full map or graph of the surrounding can be achieved.
Another benefit of using the camera is the ability to know the orientation of the camera in the world coordinate system.

This general method, of using the camera to allocate itself in a 3d coordinate system, is called state estimation in the literature.

# State estimation

Pose estimation, in the context of cameras and robotics is the task of determining the object position and orientation in 3D space, at any given moment.

State estimation is a crucial feature in autonomous robotics – self driving cars, drones, ground robots and much more. It is vital for the robot to 'know' where it is in 3D space, so it can follow direction as to where to go. Also, it is vital that it won't collide in any physical obstacle in its path.

For cameras, state estimation is used in order to create 3D content from stereo or multiple cameras constructions.

## State estimation approaches

The process of object state estimation have many implementation that range over a variety of data acquisition methods and different solving techniques.
In this paper we will focus on visual odometry (VO) – estimating the camera location and orientation in 3D space using sequential images.

Visual odometry can be separated into three different approaches,

### Feature detection based methods

Detecting and matching features from selected two frames in order to calculate the local position and orientation change between said frames.
This is the most commonly used approach for VO, it is easy to implement and robust to various noises.

Commonly used algorithm may include PTAM [2], parallel tracking and mapping, which extract FAST [3] features for tracking and simultaneously it solves the mapping task (bundle adjustment) from decoded keyframes. By combining the tasks in parallel, PTAM can achieve real time performance. PTAM [2] was created for AR scenes and small indoor spaces.

Another state of the art algorithm is ORB-SLAM [4] (and ORB-SLAM2), which its main contribution is the use of ORB features (binary features), which are faster than SIFT. Additionally, ORB-SLAM [4] outperform PTAM [2] due to the fact that it introduce loop closure to the graph, thus improving performance.

### Direct methods

Using the displacement of pixel intensity values and incorporate that data into an optical flow scheme to determine the camera state – this methods are with a probabilistic nature.

An example for such method is the LSD-SLAM [5], which uses a coarse-to-fine algorithm to align consecutive frames and thus resulting in a depth map.
Direct methods require a lot of computational power (going over all the pixels) but the innovative approach of LSD-SLAM is that it sample pixels only near the image boundaries allowing it to work in real time. A downside to that is that LSD-SLAM recovers only a semi-dense depth map.

### Inertial measurement unit (IMU)

Most robots or mobile devices have a built in inertia sensor. State estimation can benefit from this additional input, mainly for metric scale reconstruction and frames selection.

An excellent example is the VINS-MONO [1] paper, which is the main paper this project is based on.

## General VO algorithm

Most real time VO algorithms use the following scheme,

1. Detect features in image and set a feature descriptor to each feature so it can be matched later.
2. Apply image correction (to correct lens distortion)
3. Match features from different images.
4. Calculate relative rotation and translation between matched frames.
5. Solve a nonlinear problem that minimize the re-projection error of the 3D representation of each feature in all of the frames.
6. Incorporate new frames with existing 3D features and preform stages 3 through 5 again.
7. Solve the scale of the 3D estimation (In this paper implementation, through the IMU).

# Photogrammetry basics

The project at hand is a hands on, technical project. As such, it assumes that the reader has some background in computer vision and photogrammetry in particular.
In this chapter a few basic concepts will be introduced.

## Projection matrix

Projection is the act of transforming 3D world point to its corresponding 2D point on the imaging sensor.

The formulation to do so is as follows,

$$p = K \cdot [R\ |t] \cdot P$$

Where,

$P$ – 3D homogenous point.
$p$ – 2D homogenous point.
$K_{3X3}$ – The intrinsic matrix.
$R_{3X3}$ – Rotation matrix.
$t_{3X1}$- Translation vector.
$R, t$ – Are referred as extrinsic parameters.

## Intrinsic matrix

A transformation matrix unique to each camera that transforms a point before the lens to a point on the sensor.

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

$f_x, f_y$ – The focal length per axis.

$c_x, c_y$ – The central pixel position per axis.

In this project we assume square pixels without skew.

## Fundamental matrix

The fundamental matrix $F$ connects two sets of matching 2D points from different views – these sets of points are called corresponding points.
The following formulation is called the epipolar constraint,

$$x_1^T \cdot F \cdot x_2 = 0$$

Notice that $F_{3X3}$ and that the points are 2D homogeneous points.

## Essential matrix

Equivalent to the fundamental matrix, only difference is that the points are normalized by their intrinsic parameters (the K matrix).

$$x_1^T \cdot K_1^T \cdot E \cdot K_2 \cdot x_2 = 0$$

Another representation of the essential matrix is,

$$E = [t]_x R$$

Where $[\quad]_x$ is the Kronecker product.

From it we can detect the relative rotation and translation between two viewpoints (given more than 5 corresponding points) – More could be read here.

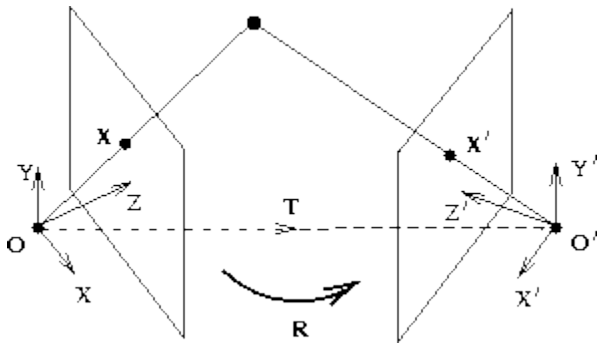



*Figure 1: Visualization of R,t using the Essential Matrix*

Given enough corresponding points, one can calculate the fundamental matrix (8 points Algorithm) or the essential matrix (5 points Algorithm – same concept as previous), usually done using RANSAC to remove outliers.

Essential matrix is used when the intrinsic parameters are known, as oppose to the fundamental matrix where they can be unknown.

## PnP problem

The PnP problem (Perspective n Points) is stated as determining the projection matrix of a given camera where the input is a set of n matching 3D points – 2D points.
More could be read here.

## Triangulation

Given a point in two images that correlate to the same 3D location and the camera matrices for each image, one can calculate the 3D location using triangulation. More could be read here.

## Reprojection error

When optimizing a camera's projection matrix a commonly used error metric is the reprojection error – i.e. the error in 2D pixels of the pixel location and the projected 3D point.

$$err = \|P \cdot Q - q\|$$

$Q, q$ – The relevant 3D and 2D points respectively.

$P$ – The Projection matrix.

# VINS-MONO

A monocular visual – inertial, odometry estimation system.
Consisting of a low cost camera and IMU to produce high quality pose estimation at minimum resources in real time.

VINS-MONO incorporates the pre-integrated IMU measurements with the visual feed from the camera to a tightly-coupled nonlinear optimization, with the addition of loop closure, failure detection, Relocalization and fast initialization – it can obtain high pose accuracy with high robustness to noise.

VINS-MONO algorithm is open sourced, and was used in this project as the ground algorithm to work with.

The following chapter will present the algorithm throughout its stages.

## VINS-MONO Algorithm overview

In the following section, the algorithm's stages will be presented and thoroughly explained.

All the figures in this section are taken from the official paper.

### Preprocessing

The preprocessing stage is per frame – for each new frame that arrives preprocessing is performed.

#### *Feature detection*

Feature detection is separated into two main steps

- Features from previous frames – all features within the sliding window are tracked using the KLT sparse optical flow algorithm.
- New features – Current frame is passed to a feature detection approach based on the paper *Good features to track* [6].
  The paper's main concept is that it track the image motion through pixel intensities. And calculate interest point based on the pixel intensity movement.

  After features are extracted they are stored as BRIEF [10] descriptors for future matching.

### *Feature matching*

Feature matching is the act of matching BRIEF [10] descriptors.
There are two types of feature matching

- 2D-3D feature matching, previously detected 3D points have a BRIEF descriptor assigned to them. Thus, we can match features in new frame to already known 3D points.
  2D-3D matching is ideal, relieve us from creating the 3D points (because they already exists) for each 2D feature.
- 2D-2D feature matching, between frames inside the sliding window – done for features that exist in new frame and were not matched before.
  2D-2D features are matched using BRIEF descriptors as well.

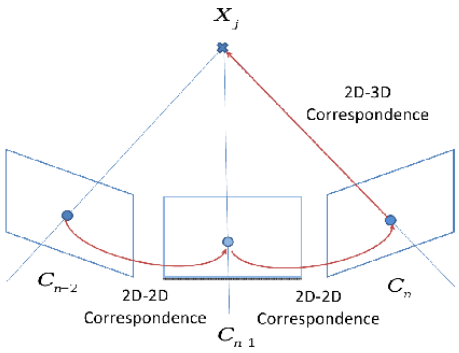
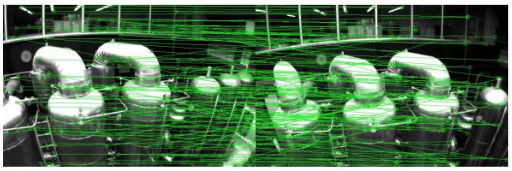



*Figure 2: Visualization of feature matching*

### *Outlier removal*

Matched features between two frames are used to calculate the fundamental matrix and outliers are filtered using the RANSAC Algorithm [7].
Outlier removal for existing feature matches (2D-3D) is also done with RANSAC, by checking outliers to the PnP problem.



(a) BRIEF descriptor matching results

(b) First step: 2D-2D outlier rejection results

(c) Second step: 3D-2D outlier rejection results.

*Figure 3: Visualization of Outlier removal*

### *Frame selection*

Frames are selected based on the average parallax from the previous keyframe, parallax can be obtained by the IMU.

Notice that IMU rotation and translation data is used to determine parallax but not the relative frame position and orientation because IMU data is known to suffer from drift error. Another criteria to determine if the current frame should be considered as a keyframe is the number of tracked features.

Both criteria have a parameter as a threshold.

## Initialization

### *Relative rotation and translation*

To recover the relative rotation and translation between two frames we use the method described in chapter "Photogrammetry basics" sub chapter "Essential matrix".

### *Triangulation*

Preform triangulation on all features between the two frames, use arbitrary scale.

### *PnP problem*

To set an initial guess to the bundle adjustment optimization (which is shared to all frames) we solve the PnP problem for all frames in the sliding window.
Meaning, for each 2D feature that has a corresponding 3D feature, we can solve the projection matrix (The matrix transformation from 3D point to 2D point on the image) of all of the frames inside the sliding window.

### *Visual bundle adjustment*

To further increase accuracy and robustness, a nonlinear least square problem is solved to minimize the reprojection error inside the sliding window.

### *Visual-inertial alignment*

Previous steps were using solely the visual data – meaning it is not up to metric scale.
By combining the pre-integrated data from the IMU the scale of the system can be determined. This is used as an initial step to the following bundle adjustment.

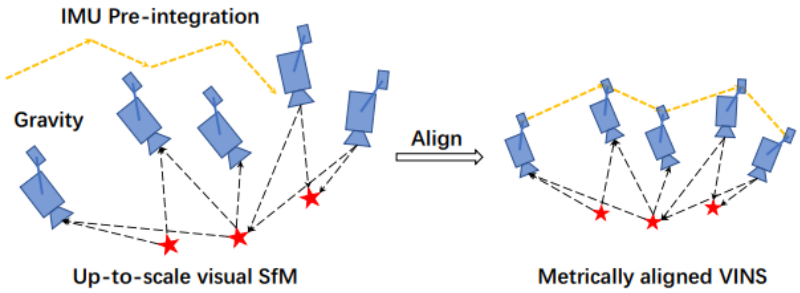



*Figure 4: Pose alignment to metric scale*

## Tightly coupled visual inertial odometry

### *Visual inertial bundle adjustment*

Applying both visual constraints (reprojection error) and IMU constraints to a single cost function, solving the nonlinear problem will achieve a maximally accurate system – inside the sliding window.

## Relocalization

Using bag of words approach (chosen implementation from this paper, DBoW2 [9]), we can find candidate keyframes to be loop closure frames. Features from candidate frames are matched and outliers are removed (same as in the above process).
Another tightly coupled nonlinear BA system is solved – adding the Relocalization term to the cost function.

After this stage the sliding window graph is connected to the entire pose graph.

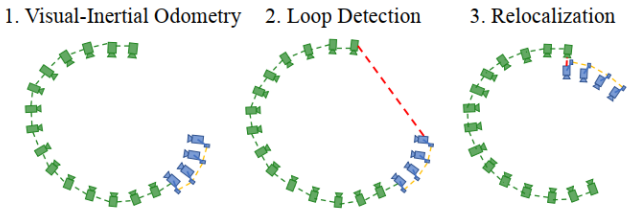



*Figure 5: Relocalization example*

## Global pose graph optimization

### *Adding new keyframes*

New keyframes in the pose graph are added and determined whether they are sequential edges or loop closure edges (later on, sequential edges are going to be removed to save memory).

### *Full bundle adjustment*

Due to the fact that the pitch and roll angles are without drift, the full bundle adjustment pose graph is optimizing only four degrees of freedom – X, Y, Z, Yaw angle.
All other variables are considered constant.

The above assumption is done to be able to solve the pose graph in real-time.

Side note, the Relocalization and Pose graph optimization are performed in separate threads. This is done to allow Relocalization to work with the current pose graph available.

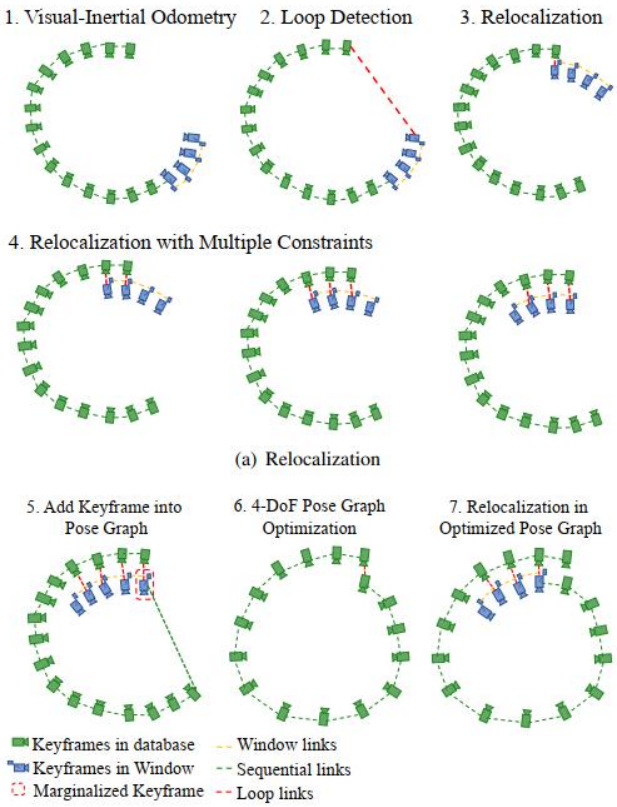
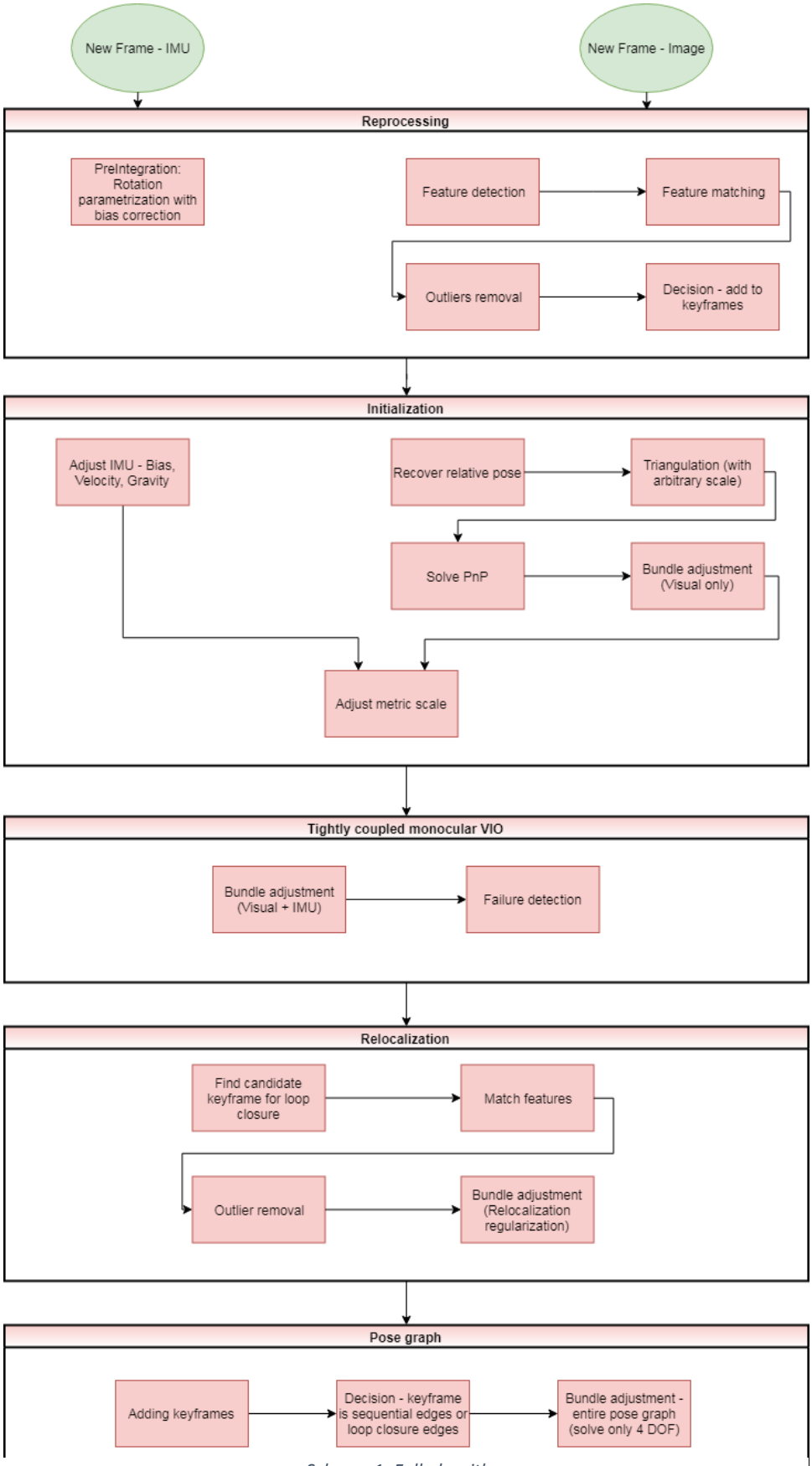



*Figure 6: Loop closure and optimization*

## Schematic algorithmic overview




*Scheme 1: Full algorithm*

## Algorithm and code relations

In this project we used the provided open source implementation of the VINS-MONO [1] algorithm.

The following chapter will present the algorithm's code and will relate it to the different algorithmic stages that were presented before.

The code is written in C++ and is wrapped with ROS (a robotic library that will be explained ahead), ROS's IO is through publishers and subscribers that pass the data from the hardware onto the application. In our case the hardware is a camera and an IMU.

The code is divided into three main blocks

### Feature tracker

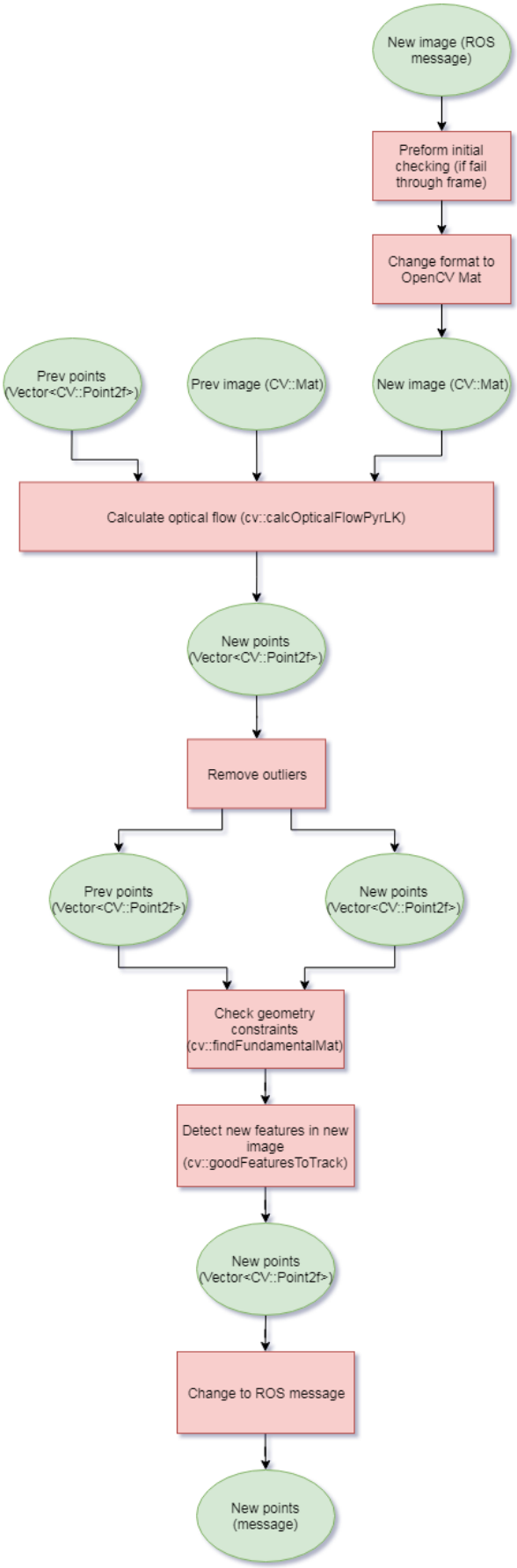
Receive new image as input and outputs a set of feature points.

The following scheme describes the algorithmic flow for the feature tracker code.

First Optical flow is calculated to match existing features, just as in *Feature detection* section in the theoretical overview.
Then, the fundamental matrix is calculated in order to remove outliers – as in the *Outlier removal* stage in the theoretical overview.

When finished processing existing features, new features are extracted as in the *Feature detection* stage.

And finally the features are transformed (in a ROS message format) to the estimator block.




*Scheme 2: Feature tracker*

## Estimator

Receives IMU data and feature points as input, and outputs the calculated pose.

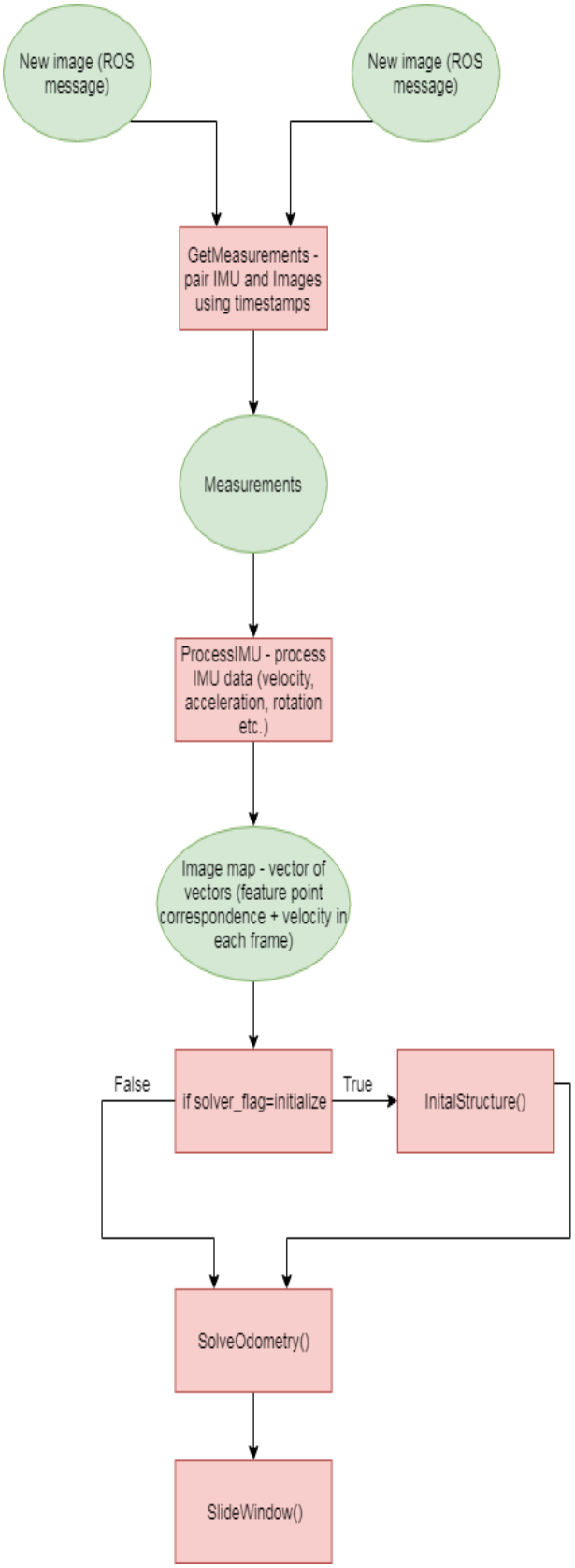
GetMeasurements pairs each image with a set of IMU readings with timestamps from previous image up to current image – this is done due to the high publish rate of the IMU. The IMU measurements are paired to the image in ProcessIMU.

These two steps can be seen as the *IMU preprocessing* step in the theoretical overview.

After that the code splits based on the current state – if this is the start of the process (solver_init = false) then an initialization must be performed. This stage holds the *Initialization* step from the theoretical overview.

If the solver has finished initialization (from previous frame or after a successful initialization in current frame) it can pass on to the next stage which is the solveOdometry – this is the main bundle adjustment for the local pose graph – its theoretical counterpart is *Tightly coupled monocular VIO.*

The SlideWindow function is just a movement function to the next frame.




*Scheme 3: Estimator*

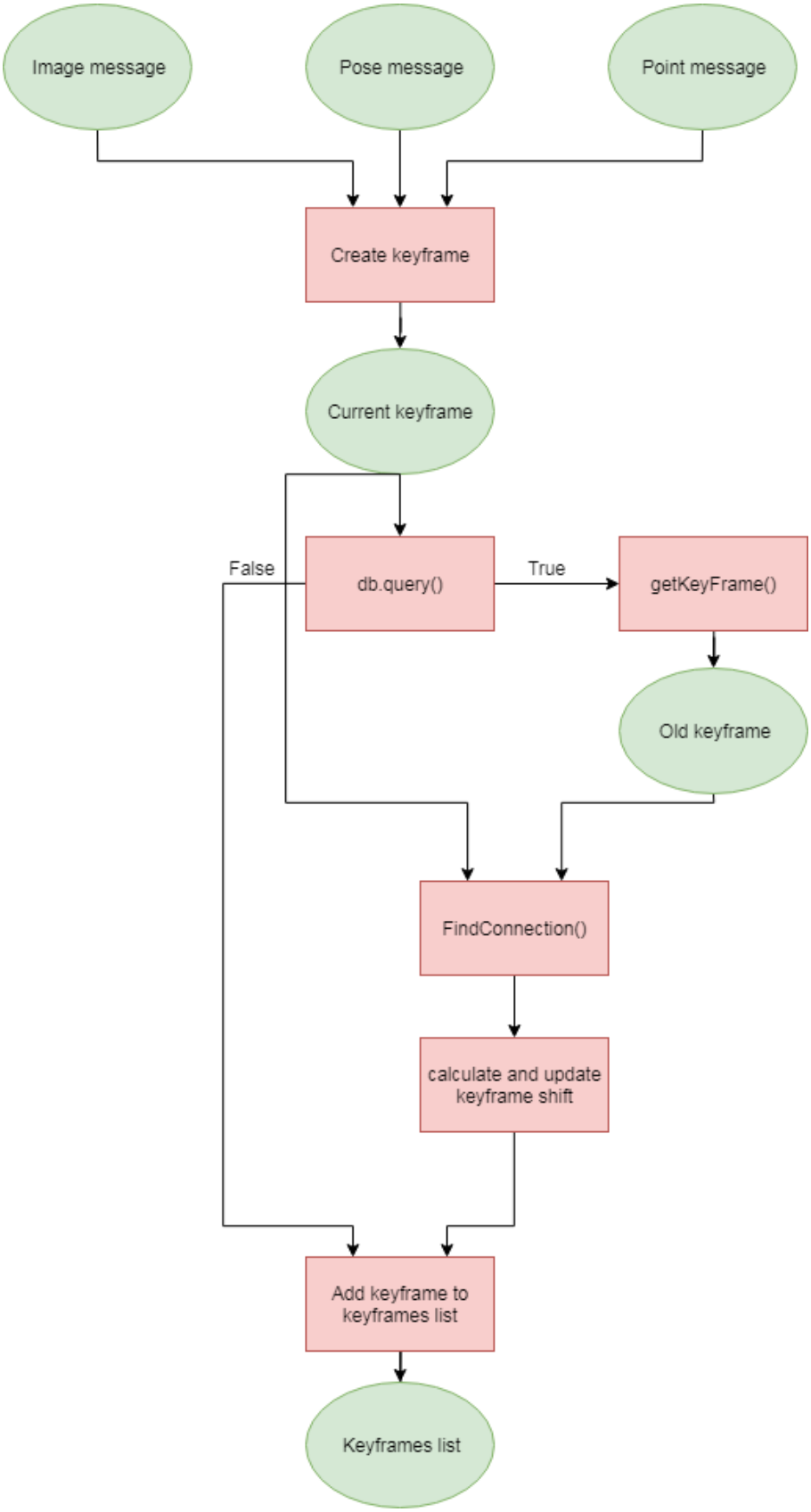
### Relocalization + Pose graph

Receives publishing from the estimator and the sensors (camera) and transform the data of the new keyframe to the keyframe class.

After several sanity checks, search for the keyframe in the database – the search is done by BRIEF [10] descriptors and only the oldest keyframe that passes a threshold is retrieved. This stage correlate to the find candidate stage in the *Relocalization* part of the theoretical algorithmic overview.

FindConnection searches and match features and remove outliers between the keyframes – as the rest of the stages in *Relocalization* in theory – the relevant shift between keyframes is calculated and adjusted.

On a different thread (not visible in the diagram) a Ceres optimizer is running in an infinite loop and optimize the entire updated pose graph – this process is matches with the *Pose graph* section.




Scheme 4: Pose graph

## VINS-MONO Results

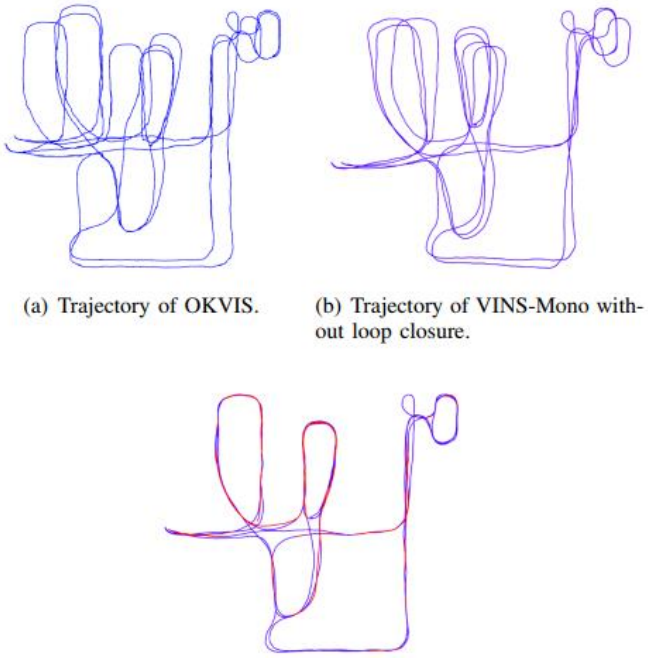
In the paper numerous tests were conducted to demonstrate the effectiveness of the algorithm for different scenarios

- Indoor scene
- Large scale outdoor scene
- Comparison to state of the art (OKVIS [11]).

Furthermore, the algorithm was tested on a mobile device to present how it can work on a low performance device and maintain real time performance.

The dataset that was chosen was the Euroc MAV Visual Inertial Dataset [12].

One can see the benefit of using VINS-Mono with the Relocalization and loop closure over state of the art algorithm,



(a) Trajectory of OKVIS.

(b) Trajectory of VINS-Mono without loop closure.

(c) Trajectory of VINS-Mono with relocalization and loop closure. Red lines indicate loop detection.

*Figure 7: VINS-Mono paper's results*

# Project preliminary work

Prior to preforming experiments with a real drone in a control setting, some preliminary work was done.

That preliminary work can be divided into several sub-categories

## Literature review

Several papers and thesis works were reviewed before deciding to use the VINS-MONO [1] paper's implementation – the goal was to find a paper that will include an innovative approach and also a free to use implementation.

## ROS Environment tutorial

Almost all implementations that include both a machine vision process and robotic interface are using ROS (Robotic Operating System).

ROS is a software framework for writing code for the purpose of robotics which can connect to hardware device controls.
ROS introduce a robust convention to writing code for robotics which its infrastructure lays on the concept of publishers, subscribers and ROS messages. Messages can be looked at as the way to transfer data from one robotic node to another, The publishers are responsible to send the ROS messages and the subscribers are in charge of starting a specified functions (called callbacks) when a message has been delivered.

ROS has many libraries and infinite possibilities to use it. The main contribution is the robust and intuitive framework which everybody can use for their own purposes.

In VINS-MONO [1] for example, the main estimation algorithm needs to receive data from the camera and the IMU. This data is transformed into ROS messages and published to the main algorithm which in return can publish the pose graph onto the screen. ROS can also be used to set the drone direction and velocity (this part is not included in this work).

In order to work fluently and be able to debug ROS, an online course was taken. The course included theoretical overview alongside with mini projects implementation to truly grasp the concept of ROS system.

http://www.theconstructsim.com/construct-learn-develop-robots-using-ros/robotigniteacademy_learnros/

*Figure 8: Ros course site*

## Code review

One of the reason VINS-MONO was chosen as the algorithm to use in this project, is the fact that the author of the paper made the algorithm code available for free use.

```
https://github.com/HKUST-Aerial-Robotics/VINS-Mono
```

*Figure 9: VINS-MONO offical GitHub*

So, before starting the experiment a fully comprehensive overview of the code was done. The purpose was to know the code to the smallest detail in order to predict and solve upcoming obstacles that one might have.

## Camera calibration

Obtaining the correct camera parameters is essential for the algorithm to perform as expected.

The process of calibrating the camera is rather simple, a pre-existing function from the OpenCV library was used.
To use the function, a calibration board must be placed facing the camera and the user must perform several movements – spread the camera field of view, preform skew, zoom in and out.

After that the algorithm outputs a set of camera parameters also known as intrinsic parameters. Those parameters include the focal length, the center pixel and the lens distortion coefficients.

These parameters are then written into a configuration file for the VINS-MONO algorithm to use.

To run the camera calibration tool run the following command,

```
rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/camera/image_raw
camera:=/camera
```

*Figure7: Camera calibration*

# Experiments procedure

The first stages of the project were,

- Understanding the theory behind state estimation in general and in VINS-MONO in particular.
- Learning how to code in ROS.

The following task was experiment with the code on a drone in real-time, this will be elaborated in this chapter.

## Setting the software

The OS of the laptop used in the project was Linux (Ubuntu 16.04) and accordingly the ROS version was ROS Kinetic.

The code for the VINS-MONO algorithm is written in C++. Eclipse was the chosen IDE to use, due to the fact that it has an intuitive debugger as opposed to command line debugging using GDB (GNU Debugger)

The Eclipse installation was not straightforward, therefore the installation steps are added here,

Follow instruction from:
https://www.youtube.com/watch?v=grlBIimWXgU
http://www.ceh-photo.de/blog/?p=899

- Download from website
- extract: tar xvfz "filename"
- install JAVA:
    - sudo add-apt-repository ppa:webupd8team/java
    - sudo apt update
    - sudo apt install oracle-java8-installer
    - sudo apt install oracle-java8-set-default
- double click on the eclipse installer that was extracted
- press OK through installation
- Make eclipse project:
    - cd ~/catkin_ws
    - catkin_make --force-cmake -G"Eclipse CDT4 - Unix Makefiles"
    - awk -f $(rospack find mk)/eclipse.awk build/.project > build/.project_with_env && mv build/.project_with_env build/.project
    - cmake src -DCMAKE_BUILD_TYPE=Debug
    - source ~/catkin_ws/deval/setup.bash
    - open eclipse (eclipse command)
    - open project: file->import->general->existing_projects...
    - select project from catkin_ws/build folder
- Build project from Eclipse
    - project->build all
- Debug from Eclipse
    - Inside the launch file you wish to debug
        - select the node you want to debug and add to the node tag: launch-prefix="xterm -e gdbserver localhost:10000"
    - Create ecllipse debug  configuration
        - Run->Debug configuration...->C/C++ remote application [double click]
        - Add the binary of the node you wish to debug
        - press apply and then debug

*Figure 10: Eclipse Install and usage*

## Additional calibration procedures

In addition to camera calibration, an attempt to calibrate the noise of the IMU was made using the Allen variance.
The results were not satisfactory so they were discarded.

https://github.com/gaowenliang/imu_utils

*Figure 11: IMU Noise estimation GitHub*

Furthermore, an attempt to calibrate the extrinsic parameters (translation and rotation) between the camera and the IMU was made using the Kalibr toolbox.
The results were also not satisfactory so they were discarded.

https://github.com/ethz-asl/kalibr

*Figure 12: Kalibr toolbox GitHub*

## Running VINS-MONO

To run the VINS-MONO code, preform the following steps

- Adjust configuration file to personal settings
    - IMU and camera publishers
    - Camera intrinsic parameters.
    - Camera and IMU extrinsic parameters
    - IMU noise estimation parameters.
    - Further algorithmic parameters.

A configuration file example (From the official GitHub),

```
#common parameters
imu_topic: "/imu0"
image_topic: "/cam0/image_raw"
output_path: "/home/tony-ws1/output"
#camera calibration
model_type: PINHOLE
camera_name: camera
image_width: 752
image_height: 480
distortion_parameters:
   k1: -2.917e-01
   k2: 8.228e-02
   p1: 5.333e-05
   p2: -1.578e-04
projection_parameters:
   fx: 4.616e+02
   fy: 4.603e+02
   cx: 3.630e+02
   cy: 2.481e+02
 #Extrinsic parameter between IMU and Camera.
estimate_extrinsic: 0
extrinsicRotation: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [0.014, -0.999, 0.00,
          ,0.0257, 0.0149, 0.999
          ,0.999 ,0.003, 0.025]
#Translation from camera frame to imu frame, imu^T_cam
extrinsicTranslation: !!opencv-matrix
   rows: 3
   cols: 1
   dt: d
   data: [-0.021,-0.064, 0.0098]
```

```
# feature traker paprameters
max_cnt: 150            # max feature number in feature tracking
min_dist: 30            # min distance between two features
freq: 10                # frequency (Hz) of publish tracking result. At least 10Hz for good estimation. If set 0, the frequency will be
#same as raw image
F_threshold: 1.0        # RANSAC threshold [pixel]
show_track: 1           # publish tracking image as topic
equalize: 1             # if image is too dark or light, trun on equalize to find enough features
fisheye: 0              # if using fisheye, turn on it. A circle mask will be loaded to remove edge noisy points


#optimization parameters
max_solver_time: 0.04  # max solver iteration time [ms], to guarantee real time
max_num_iterations: 8   # max solver iterations, to guarantee real time
keyframe_parallax: 10.0 # keyframe selection threshold [pixel]

#imu parameters       The more accurate parameters you provide, the better performance
acc_n: 0.08          # accelerometer measurement noise standard deviation. #0.2   0.04
gyr_n: 0.004         # gyroscope measurement noise standard deviation.     #0.05  0.004
acc_w: 0.00004       # accelerometer bias random work noise standard deviation.  #0.02
gyr_w: 2.0e-6        # gyroscope bias random work noise standard deviation.     #4.0e-5
g_norm: 9.81007      # gravity magnitude

#loop closure parameters
loop_closure: 1                 # start loop closure
load_previous_pose_graph: 0       # load and reuse previous pose graph; load from 'pose_graph_save_path'
fast_relocalization: 0           # useful in real-time and large project
pose_graph_save_path: "/home/tony-ws1/output/pose_graph/" # save and load path

#unsynchronization parameters
estimate_td: 0                  # online estimate time offset between camera and imu
td: 0.0                         # initial value of time offset[sec]. read image clock + td = real image clock [IMU Clock]

#rolling shutter parameters
rolling_shutter: 0              # 0: global shutter camera, 1: rolling shutter camera
rolling_shutter_tr: 0            # [Sec] rolling shutter read out time per frame (from data sheet .(

#visualization parameters
save_image: 1                # save image in pose graph for visualization purpose; you can close this function by setting 0
visualize_imu_forward: 0       # output imu forward propagation to achieve low latency and high frequency results
visualize_camera_size: 0.4      # size of camera marker in RVIZ
```

*Figure 13: Configuration file example*

- Publish Camera onto ROS.

```
roslaunch video_stream_opencv camera.launch video_stream_provider:=1
```

- Publish IMU onto ROS.

```
roslaunch crazyflie_demo external_position_Realsense_and_OnBoardCamera.launch
```

- Run the VINS-MONO launch file (after changing it to load previously defined configuration file).

```
roslaunch vins_estimator personal.launch
```

- Open RVIZ to view progress in real time.

```
roslaunch vins_estimator vins_rviz.launch
```

- To run a ROSBAG skip the camera and IMU publishing and publish a prerecorded ROSBAG

```
rosbag play <bagfile_path>
```

## Obstacles and solutions

During the experiments a few obstacles arose, the goal of this sub-section is to help researchers identify their problems and solve them.

- **Problem:** VINS-MONO is waiting for images and IMU data.
  **Solution:** check the topics exists,

  ```
  rostopic list
  ```

  Check that the images are published correctly - Sometime, the published images are of the laptop internal camera.
  Command line to publish the image_raw topic into the screen,

  ```
  rosrun image_view image_view image:=/camera/image_raw
  ```

  Check the IMU data is published correctly,

  ```
  rostopic echo crazyflie7/imu
  ```

- **Problem:** Eclipse debugger doesn't work\ eclipse doesn't compile.
  **Solution:** Eclipse doesn't pass compilation but ROS catkin_make does, if that is the case – numerous attempts were made to solve this issue – all were futile.
  The solution proposed is to add prints to the screen at every block and zeroing down onto the failure cause.

- **Problem:** the solution suffer from drift.
  **Solution:** this can be caused by several problems.
  The first is miscalculating the extrinsic parameters between the camera and the IMU, this have huge impact on the algorithm output. Check the following link with examples of setups and the correct translation rotation.

  https://github.com/HKUST-Aerial-Robotics/VINS-Mono/blob/master/config/extrinsic_parameter_example.pdf

  Another reason is non reported time delay between IMU and image messages, if the messages are not synced the code will not work – need to adjust your hardware so that they will.
  Finally, perhaps some other parameter in the configuration file is misconfigured – this is very subjective to the camera and scene settings.

  A remark, the first two are a "self-correction" option through the configuration file – '*estimate_td', 'extrinsic = 2'*. In our experience, they both don't converge and were turned off.

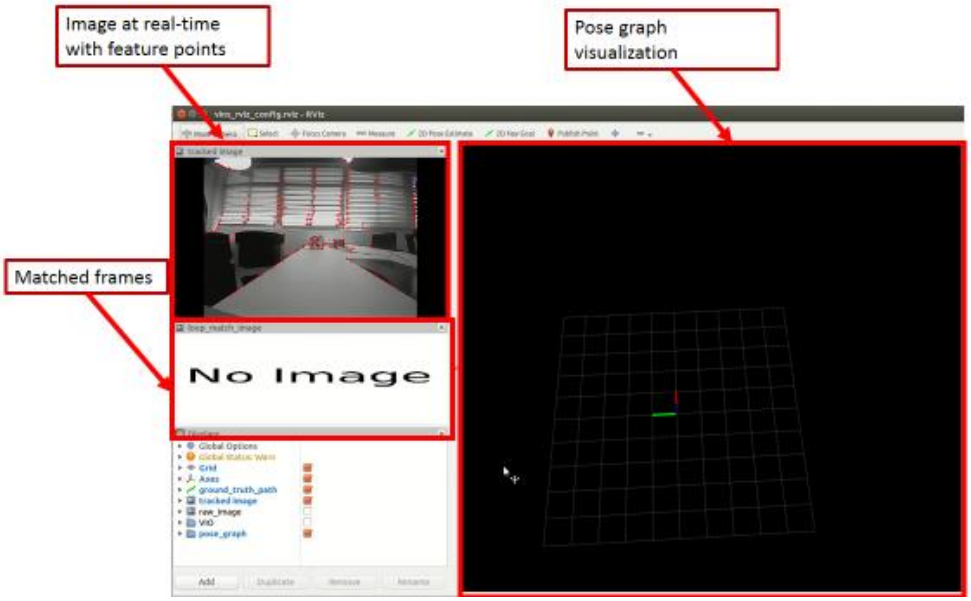
- **Problem:** the solution suffer from drift while motionless
  **Solution:** this is a known issue with VINS-MONO (according to the forum thread in the GITHUB). Still, the solution that worked for us, is to fasten the camera to the IMU in a more robust manner which correlate to having an error in the extrinsic parameters.

- **Problem:** Relocalization not working – i.e. same physical location appear at tow location on the graph.
  **Solution:** This happen when the features from the newly acquired frame aren't match to any previous frame.
  The cause could be, different lightning or occlusions, try moving slightly in the same place.
  Another cause, the previous frame from that location is to old – it was removed from the dictionary database.

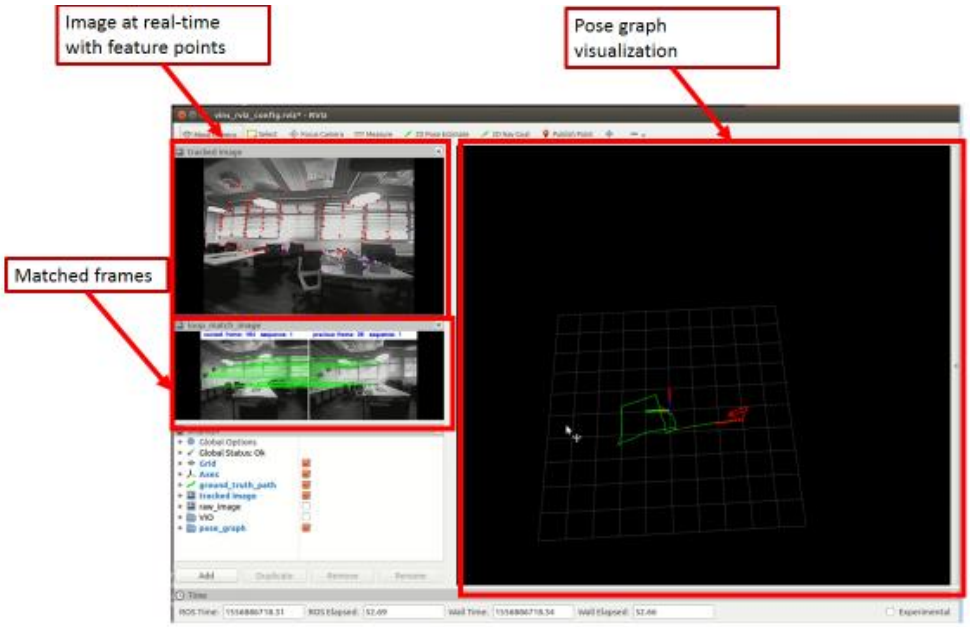
## Experiments results

First frame visualization in RVIZ – After running all the scripts as described before,

For the first frame, it is clear that the pose graph is empty and so is the matched frame window.




Loop closures: Here it is visible that a previous frame was matched and authentication to that can be seen in the pose graph – the same frame has been "visited" (random frame capture).



# Summary

This project examined the task of drone state estimation using a monocular camera.

At first, the subject of state estimation on its various approaches was researched.
An algorithm was chosen to match our criteria, VINS-Mono.

The VINS-Mono algorithm was presented in the project book – both as theoretical overview from the article and as hands on code review from the official GitHub.
Later on the paper's results were presented to acknowledge the choice of said algorithm.

The project consist of an experiment that was performed. The preliminary work is presented – all the software installations and coding languages that take place in the code (C++, ROS).

After that, the actual experiment was conducted and it results are also presented.

Finally, to support future generation to use VINS-Mono algorithm a "problem solving" section was written, with specific issues that arose during the setup and the run of the algorithm and our proposed solution to said problems.

# Reference


[1] Tong Qin, Peiliang Li, and Shaojie Shen. 2017
VINS-Mono: A Robust and Versatile Monocular Visual-Inertial State Estimator.

[2] Georg Klein, David Murray. 2007
Parallel Tracking and Mapping for Small AR Workspaces

[3] E. Rosten and T. Drummond. 2006
Machine learning for high-speed corner detection

[4] Raúl Mur-Artal, J. M. M. Montiel, Juan D. Tardós. 2015
ORB-SLAM: A Versatile and Accurate Monocular SLAM System

[5] Jakob Engel, Thomas Schöps, Daniel Cremers. 2014
LSD-SLAM: Large-Scale Direct Monocular SLAM

[6] Shi, Jianbo, Tomasi, Carlo 1993.
Good Features to Track

[7] Martin A. Fischler, Robert C. Bolles. 1981
Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography

[8] Richard Hartley, Andrew Zisserman. 2003 (Second edition)
Multiple view Geometry in Computer Vision

[9] D. Galvez-Lopez, J. D. Tardos. 2012
Bags of binary words for fast place recognition in image sequences

[10] Michael Calonder, Vincent Lepetit, Mustafa Ozuysal et al. 2012
BRIEF: Computing a Local Binary Descriptor Very Fast

[11] S. Leutenegger, P. Furgale, V. Rabaud, et al.
Keyframe-Based Visual-Inertial SLAM using Nonlinear Optimization.

[12] Euroc MAV Visual Inertial Dataset