A project submitted toward the degree of

Master of Science in Electrical and Electronic Engineering

Applying SLAM algorithm on low-cost common hardware with camera and IMU

by

Andrii Ahranovych

This project was carried out in School of Electrical Engineering

Under the supervision of Mr. Yonatan Mandel and Prof. Ben-Zion Bobrovsky

August 2021

Abstract

Drones around us becoming a popular scene, with higher demand for autonomation transportation, surveillance, and delivery systems. But at the same time there is still work to be done until full autonomation of such system will be common, safe, and legal to be used freely. With this said, plenty of pioneering work is being done in the field, to overcome different limitation that not yet been solved in different aspects.

In this work we will show proof of concept applying ORB-SLAM3, a new improved open-source SLAM algorithm, integrated into low-cost remote hardware which stream synchronised inertial and image data streams through ROS framework into host pc over the air. We will also apply VINS-MONO on same setup to show how easy it is to change the algorithm without changing the setup.

ORB-SLAM3 and VINS-MONO are real-time SLAM libraries capable of performing, Visual-Inertial and Multi-Map SLAM algorithm with monocular, or stereo and RGB-D cameras for ORB-SLAM3, using pin-hole and fisheye lens models. We will work with pin-hole model and a single M12 regular lens model.

We will detail the steps to reproduce our work, and account limitations and suggest future work. This project is about showing that autonomous system doesn't require high end technology and can be constructed by anyone with common hardware.

Contents

Table of figures

1. Project Setup

The project consists of the following hardware and software components:

Hardware we used:

1) SSB computer - Raspberry Pi 4 B
2) Global shutter camera - Arducam UC-599
3) Inertial measurement unit – we used MPU9265
4) Wi-Fi 2.4GHz router – optional
5) Host PC with i7 cpu – we used Gigabyte GB-BRi7H-8550 mini PC

Software and frameworks:

1) Raspbian OS installed on RaspberryPi
2) Ubuntu 18.04 installed on host PC
3) ROS melodic installed both on Pi and Host PC
4) Arduacm camera driver installed on Pi
5) ORBSLAM3, Kalibr installed on host PC

Block Diagram of our setup:

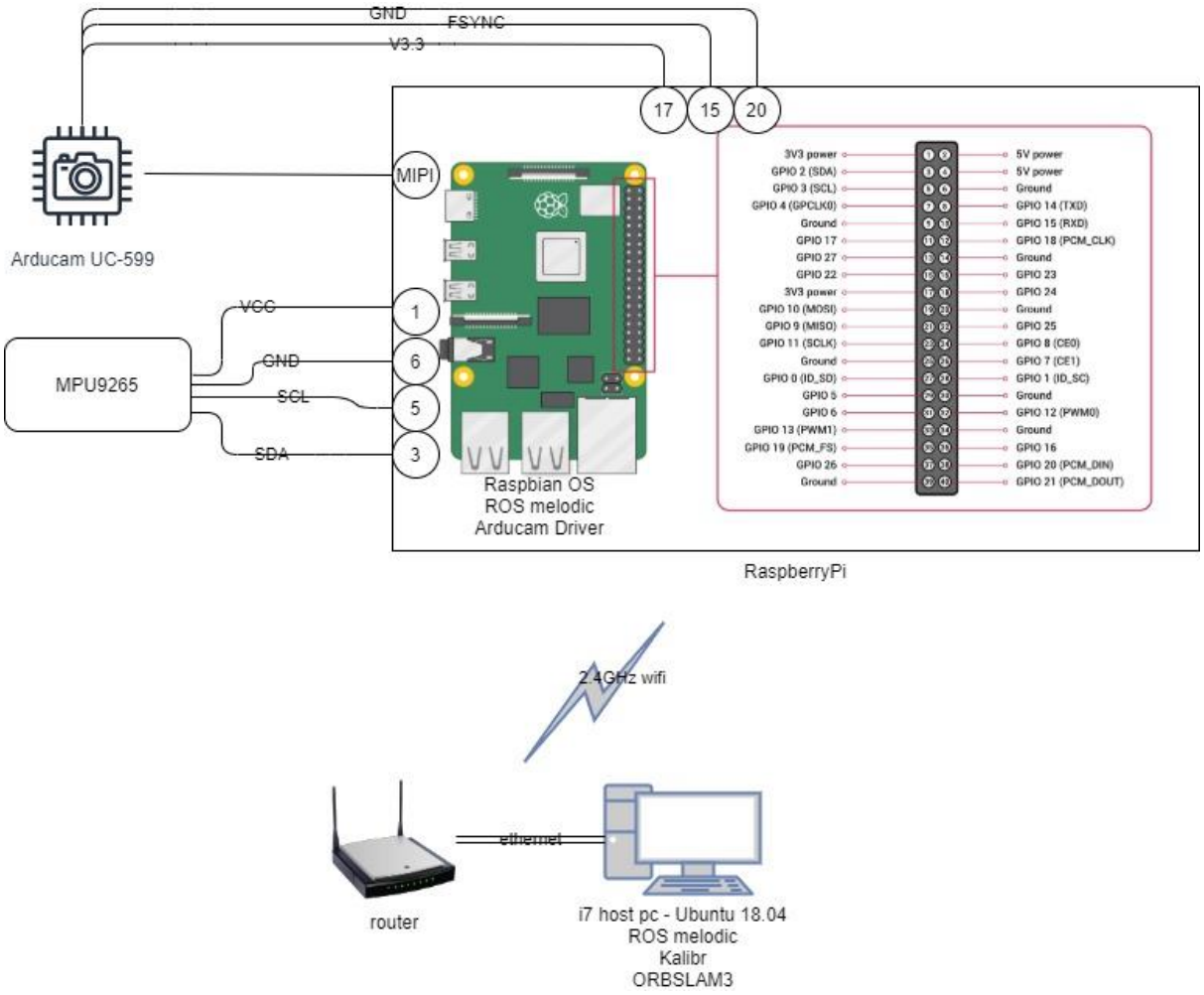

Figure 1- connection block diagram of setup

The diagram above shows port connections we used in our design. The text written above connection lines represent the pin naming which appear on IMU and camera boards. The circles with numbers in them represent GPIO port used on Pi's board.

As it can be seen, camera module is connected via two types of connectors to Pi's board. MIPI connector is bus connector which responsible for data transport from camera module to Pi and module control by Pi. Where is, FSYNC port that connects to GPIO ports is responsible for picture capture through external signal, in our case it will be generated by Pi.

The IMU sensor connects directly to Pi GPIO ports and in our scenario uses I2C communication protocol.

To hold components together we used 3D printed plastic casing on which we mounted Pi, IMU and camera together as whole system, this represents a future drone system. The casing model:

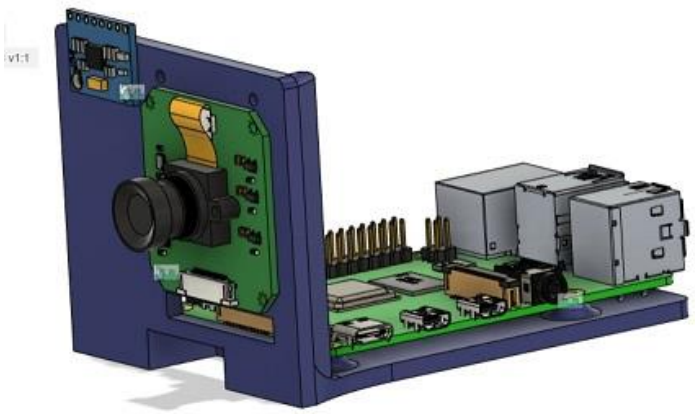

Figure 2 - mounted system computer model

Bellow can be seen fully assembled system photos:

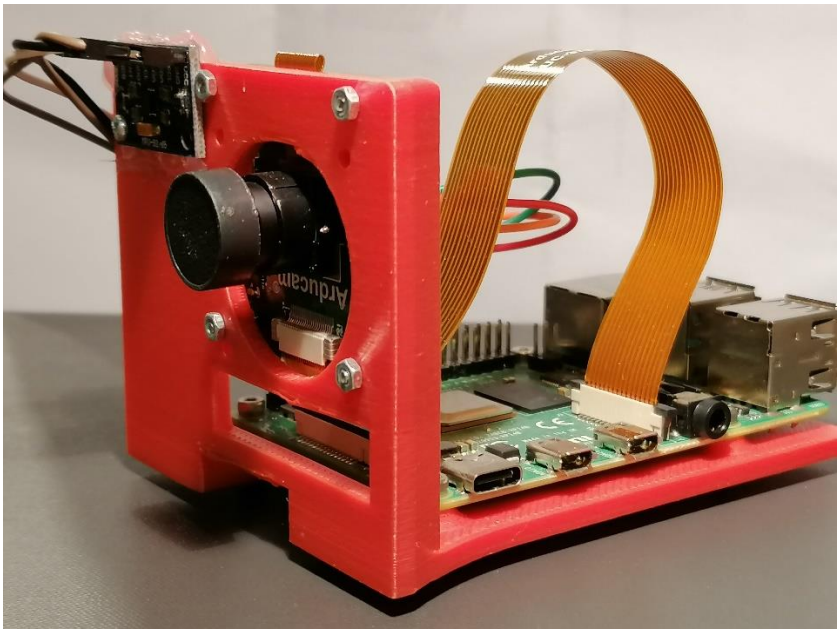

Figure 3 - side view of assembled system

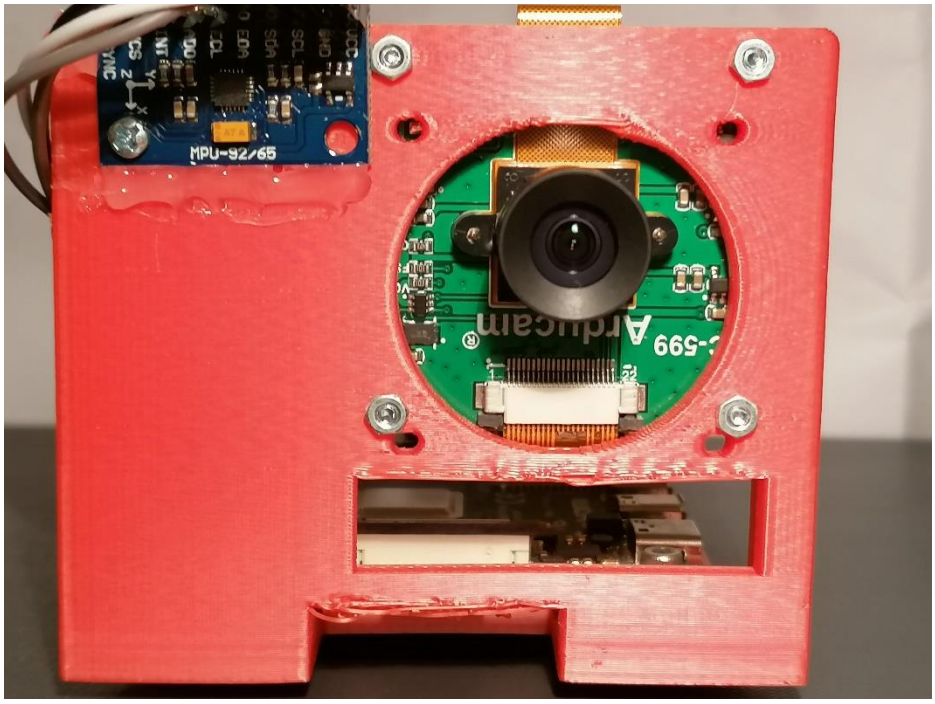

Figure 4- rear view of assembled system

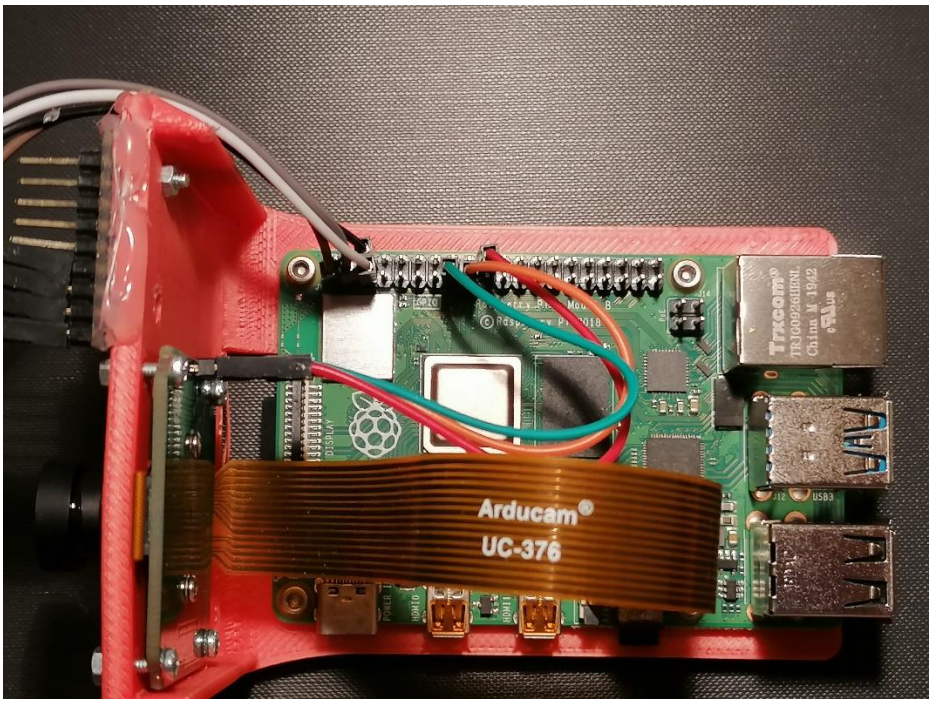

Figure 5 - top view of assembled system

We will now look at the ROS framework flow we used in the project:

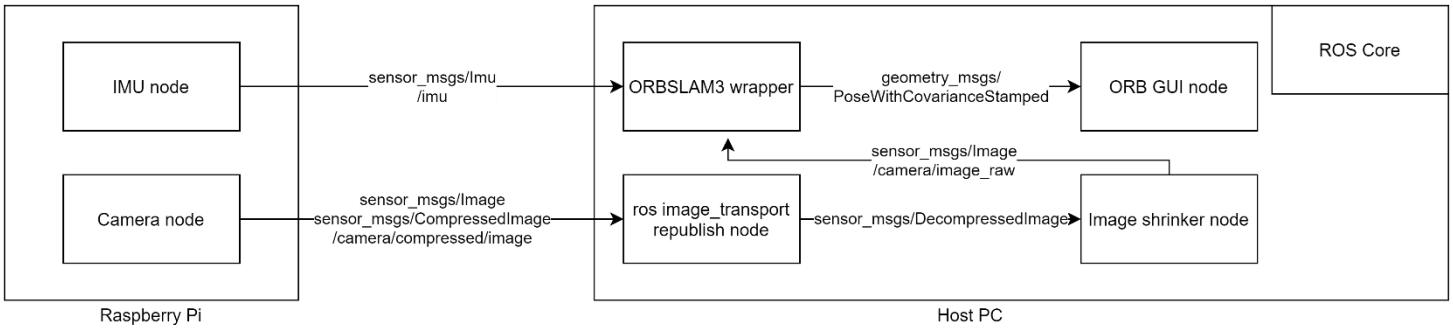

Figure 6 - ROS node and topic flow in system

As can be seen in the above diagram, our system passes two type of sensor data messages, IMU message and image message, where is due to limitation of bandwidth passing through 2.4GHz type N Wi-Fi protocol, in real, transmitting saturated, environment we were unable to achieve constant 20 fps 1280x800 resolution image stream, therefore we used built ROS compression image node, known as image_transport, and forward a constant 20 fps JPG compressed image stream. ORBSLAM is incapable of working with compressed image message, therefore we use at host side republishing topic responsible for decompressing back to raw image. Then, topic is published to image binning node. Image binning node is required if one willing to work with lower resolution image due to computational hardware throughput limitation required by ORB-SLAM3 algorithm. In our setup we shrunk resolution from 1280x800 to 640x400.

We will now detail the different components used in this project.

2. Arducam UC-599

We used this camera module as it comes with corresponding driver source code, and an integrated support inside Pi's OS kernel. We chose to use global shutter camera over rolling shutter, as rolling shutter cameras suffer from spatial distortion and so it will negatively affect performance of SLAM algorithms.

The integrated CMOS image sensor used in this model is OV9281. OV9281 is a 1-megapixel, black and white sensor, allowing capture of images at maximum of 120 fps at maximum resolution of 1280x800 pixels. The output interface is through 2-lane MIPI serial output which connects directly to RPi board as seen in the diagram (1).

Figure 7 - UC-599 model front view

The model comes with M12 lens with FOV of 130 degrees. It is imported to note that the lenses are interchangeable with any lens from M12 series. Pixel size of this sensor is $3\mu mX3\mu m$ and optical size of ¼ inch.

UC-599 exposes the OV9281 interface for external trigger image capture via 3 connectors as seen in diagram (1). We used this interface to allow synchronization between readings from IMU sensor and gathered image.

To use this module easily we used the driver provided by the company at their Github repository - https://github.com/ArduCAM/MIPI_Camera. We used the RPI driver for MIPI camera modules, instruction for installation can be found inside the repository.

Afterwards we wrote a ROS node using OpenCV and Arducam's driver that is configured to mode number 13 as described in the repository and was confirmed by writing out camera working modes, this is the sole operation mode that enables external trigger operation mode. This mode provides image of 1280x800 and for this driver release didn't allow any other resolution to be set.

The ROS node that was written is called, cpp_stream – as it was written in C++, and it exposes 2 parameters that could be set for camera configuration at launch time of the node:

- gain – a value between 1 – 15 that defines camera gain value that used in the A2D converter inside the sensor. Where 1 defines no gain, which for dark environment will lead to dark images, and 15 defines maximal gain which in dark environment will lead to white saturated image.
- exposure – a value from 0 to 65535, represents amount of exposure time of sensor before image is captured and transported. The larger exposure time the brighter the image will be, but with higher saturation and less sharpness, for good results at high rates of exposure we should work in still environment which is impractical for drone applications. Moreover, the higher the exposure rate the lower effective FPS that is achievable. In our case for indoor usage with different environment lighting the range of exposure rate which were practical to use was between 300 up to 1000.

Each of these values have a default setting used by the driver when virtual sensor object is constructed. Default value could be written out by reading register value, this capability is embedded into node's topic. The code also includes FPS feature which is unused under external trigger setting.

The topic that is being published is of type sensor_msgs::Image/CompressedImage. Both topics are published thanks to image_transport package that we used. Reasoning for this use is due to bandwidth limitation at transport level of HTTP messages between the remote and host. A single raw mono 8-bit image at 1280x800 resolution will require approximately 1MB which are 8-Mbits per second of constant data flow. Using visual SLAM algorithms require good FPS for feature tracking between frames, we used recommended FPS as was documented in ORB-SLAM3 repository of 20 FPS. Thus, the required constant flow of data is at least 160Mbit/s as we use HTTP TCP over Wi-Fi 2.4GHz which is capable of maximal 300Mbit/s throughput this was impossible to achieve. To over come this limitation, we used image_transport package which automatically allow publication of raw and compressed image topics. Where compressed image topic uses JPEG compression, it was done efficiently and allowed constant 20 FPS data stream at maximal resolution. Using such compression required bandwidth of only about 4MB/s. It important to note that time rags passed inside a ROS topic message were unchanged after first set.

At host side we decompressed the image back to raw image to be used by ORB-SLAM.

An attempt to allow exposure and gain change on the fly caused serious degradation in performance and was removed from use in cameras topic. If one desires to add such support future work is required.

3. IMU - MPU 9265

The inertial measurement unit we use has 9-DOF of which we only use 6. It uses standard communication I^2C and SPI. We used the I^2C configuration.

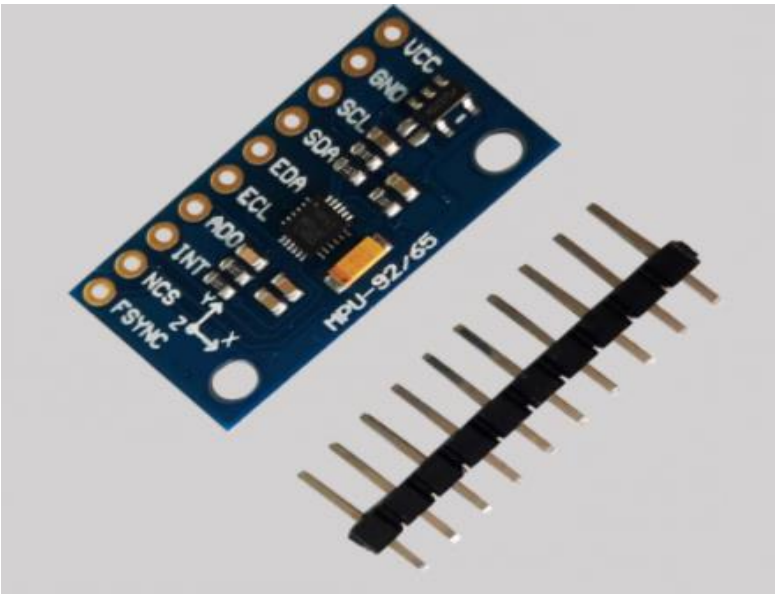

Figure 8 - MPU-9265 module

As mentioned above we only need 6-DOF which represent the angular velocity and linear acceleration. The axis system of 6-DOF is represented as follow:

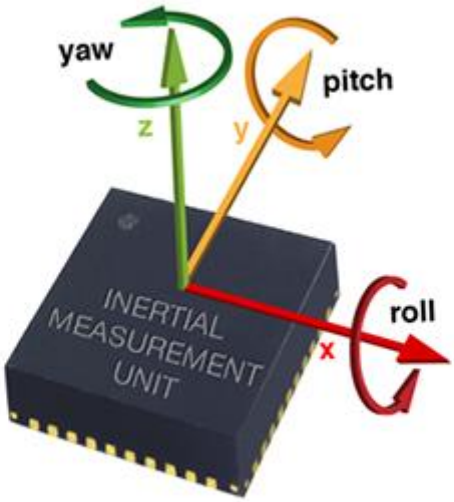

Figure 9 - 6-DOF coordination system representation

The IMU sensor is capable to achieve 400KHz communication rate and achieve sensitivity of 2000 deg/sec and accelerometer with full scale range up to $\pm 16g$.

We use configuration with maximum range for the 6-DOF at working rate of 120Hz, without any filter.

The reading from sensor registers is done through I2C interface which we connect to as seen in diagram (1). To use GPIO interfaces in RPI an instance of GPIO server should be used or other configuration method. We chose to use accessing method through Python server pigpio library. We used previously written ROS node that publishes sensor_msgs::Imu message topics and triggers camera capture signal for cpp_stream node of the camera. The repository for MPU ROS node can be found in the tau-adl repository by the name mpu9150, this sensor family are almost identical, and no changes required to work with 9265 version.

As mentioned above the IMU ROS topic is responsible to synchronise image capturing and publishing node. It is done by generating raising high signal in the corresponding GPIO pin as mentioned in diagram (1), every 6th sample cycle which for operation node rate of 120Hz will generate corresponding 20Hz rate node. It is important to note that there is difference in pin numeration between the pin numbering on the board connection diagram as seen in diagram (1) and the corresponding logical numeration of the microprocessor which is 22 in our case.

4. Raspberry Pi installation

In this project we chose to use RPI as the remote computer. The choice to use RPI is due to:

- it is relatively small – 8.5cm x 5.6cm
- it has connection to MIPI camera
- it has integrated Wi-Fi chip
- it doesn't require active or passive cooling
- it is common hardware with vast support
- it run Linux distribution

Other option we tried to work with before but had to change is the Jetson Nano development kit by Nvidia. The main reason we had to switch is because of lack of integrated driver support for global shutter camera sensor OV9281 we used. Jetson Nano had only native support to IMX219 rolling shutter sensor, and on the other hand the Arducam driver for Jetson family was through mediator driver board which we didn't purchased.

Important note regarding RPI eco-system, the native support for OV9281 is integrated into Raspbian OS only and not into Ubuntu distributions, therefore we used Raspbian OS.

To install Raspbian OS one can find instructions on official RPI website. We used the RPI in headless mode, if one desires to configure RPI in such mode, one can find instructions here: https://www.tomshardware.com/reviews/raspberry-pi-headless-setup-how-to,6028.html

After login in and updating system we installed ROS melodic distribution, it is important to note that unlike Ubuntu flavoured distribution there is no package-based distribution for Raspbian. To install ROS melodic, we followed instructions found here:

http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Melodic%20on%20the%20Raspberry%20Pi

Important note regarding the installation, there is compatibility problem with some packages in case full installation is applied. The issue occurs with different boost library versions. Therefore, we

installed only part of the available packages up to and including image_transport, by running the following line of code:

```
rosinstall_generator image_transport --rosdistro melodic --deps --wet-only
> melodic-image_transport-wet.rosinstall
```

This will generate installation list of all required packages and afterwards by following the instructions you can download, compile, and install.

Then, install the Arducam MIPI driver which is found in the official git repository: https://github.com/ArduCAM/MIPI_Camera.

After that, IMU node which can be found on tau-adl under following URL: https://github.com/tau-adl/mpu9150. Please remember to modify the topic name and GPIO pin that is used on RPI and the rate at which node must run and rate at which it triggers sync signal on camera module.

Last, install camera node we wrote which can be found once again on tau-adl under following URL: https://github.com/tau-adl/Arducam_RPi_ROS_node

5. Host PC installation

For our setup we used Ubuntu 18.04 version with fully installed desktop version of ROS melodic, instructions could be found on the following URL: http://wiki.ros.org/melodic/Installation/Ubuntu

After that install Kalibr project for system calibrations, instructions can be found here: https://github.com/ethz-asl/kalibr, this toolbox will allow us calibrate camera and inertial-camera setup.

Install calibration ROS node that calculates the error bias of IMU sensor. Installation is found here URL: https://github.com/gaowenliang/imu_utils.

Install ORB-SLAM3 package following instructions found inside projects git at following URL: https://github.com/UZ-SLAMLab/ORB_SLAM3.

Compile binning node created dedicated for reducing resolution of image from 1280x800 pixels to 640x400 pixels. ROS package code can be found URL: https://github.com/tau-adl/image_binning_ros_node.

Compile ROS wrapper for ORB-SLAM3 as the wrapper provided by the developers had a bug which prevent online running.

Install VINS-MONO package from developer's git page found at URL: https://github.com/HKUST-Aerial-Robotics/VINS-Mono. It is important to note that mismatching the versions of ceras and eigen packages will cause the algorithm to fail.

6. Calibrating Setup

After installation is done, we calibrate system by 3 steps:

1) calibrate intrinsic parameters of camera using Kalibr
2) calibrate IMU gyroscope and accelerometer bias and white noise using imu_utils
3) calibrate the whole system camera and IMU using Kalibr to find transformation and rotation between IMU and camera.

For step (1) calibration we used following calibration target, found on Kalibr github, Aprilgrid 6x6:

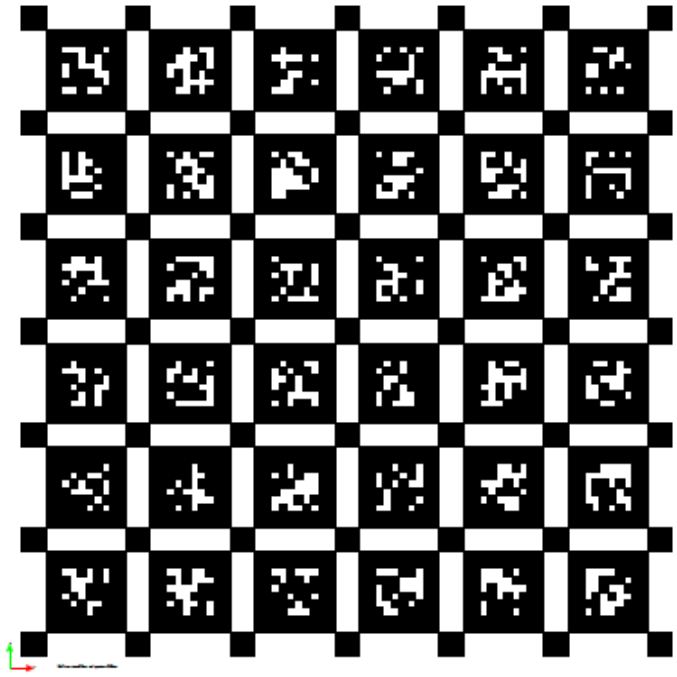

Figure 10 - calibration target Aprilgrid 6x6

We used pinhole camera model with equidistant distortion coefficients model. Our calibration target was printed on A4 paper with tags of size 2.11cm x 2.11cm and distance between tags of 0.61cm.

We made recording of camera raw image stream after decompressing it and pass-through binning node that output a 640x400 black and white raw image at 20 fps. Recording last about 60 seconds and included as vast movement as possible to fill sensor area, including twists.

Calibration results are required to be lower in error then 1px. We achieved following results:

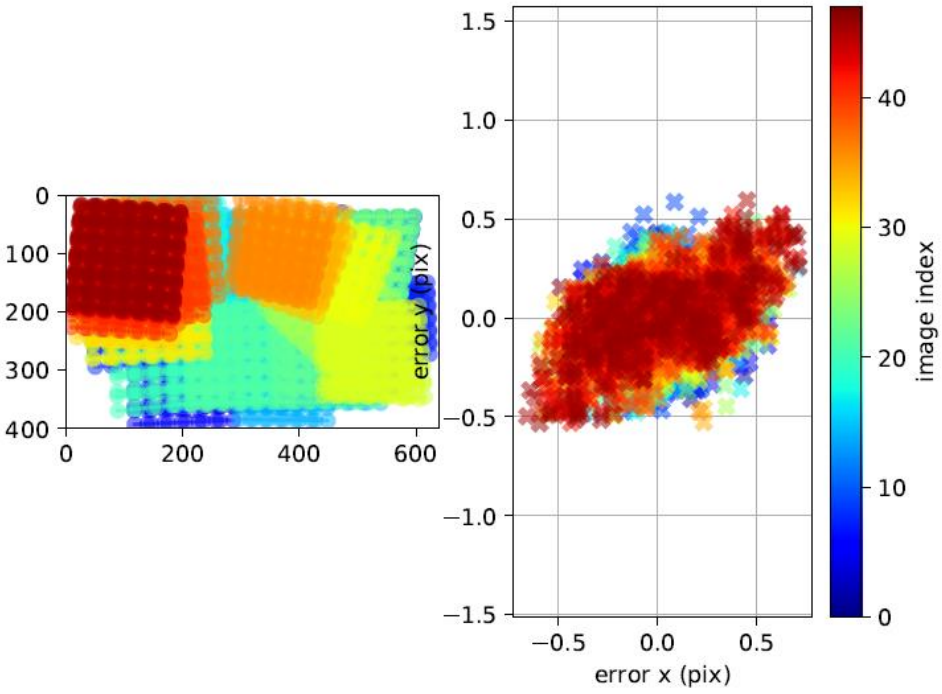

Figure 11 - UC-599 reprojection errors for 640x400 resolution

It seen that calibration reprojections errors are in desired bounds while covering most of sensor's area.

Calibration results for intrinsic parameters for our model under 640x400 resolution were:

```
cam0:
 cam_overlaps: []
 camera_model: pinhole
 distortion_coeffs: [0.31825982073706915, 0.2709370468036092, -0.22108653885390658,
  -0.1814796696573241]
 distortion_model: equidistant
 intrinsics: [443.3541154029618, 442.7629599414441, 329.24635516913645, 229.9852171160046]
 resolution: [640, 400]
 rostopic: /camera/image_raw
```

Next, we calibrate IMU sensor by streaming IMU messages on a static surface for at least 120 minutes to receive information about sensor drifting in noise and bias over time. Topic rate was set to work at 120Hz. Calibration results were:

```
#Accelerometers
accelerometer_noise_density: 6.8357826236509281e-02   #Noise density (continuous-time)
accelerometer_random_walk:   9.9062482433258793e-04   #Bias random walk

#Gyroscopes
gyroscope_noise_density:     3.4424472128302748e-02   #Noise density (continuous-time)
gyroscope_random_walk:       2.7533304166450236e-04   #Bias random walk

rostopic:                /imu      #the IMU ROS topic
update_rate:             120.0     #Hz (for discretization of the values above)
```

Last, we perform full system calibration, using Kalibr node once again. Our target is the same as was used above. We record another ROS bag this time for IMU and camera topics simultaneously and performing different movements as suggested by instruction video on Kalibr repository. Movements suggested to separate each of 6-DOF we ask to calibrate. Results we acquire are:

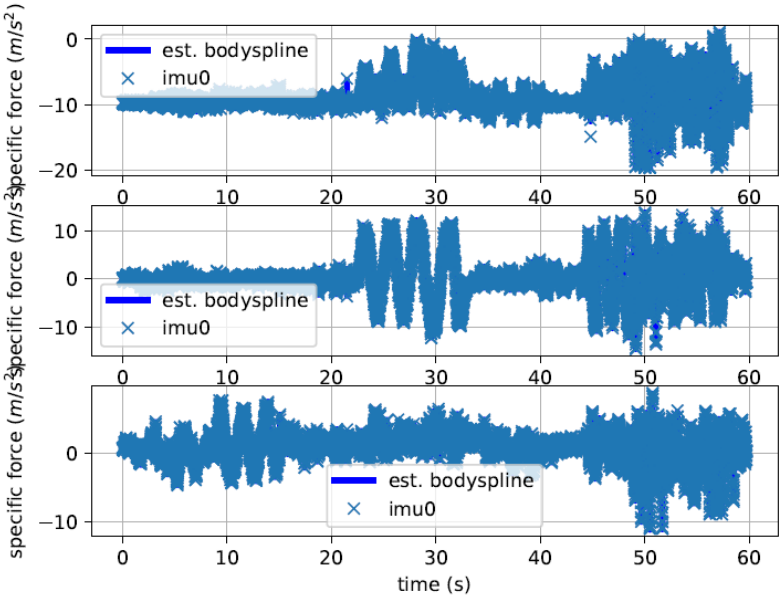

Figure 12 - comparison of predicted and measured specific force (IMU frame)

It could be seen that there is no visible difference between estimated body spline and IMU at all in all 3 axes, which is good. It is noticeable the different steps of calibration, at the beginning movement was separated to different axis with overlap for X and Y axis, but later when made abstract movements they mix all together.

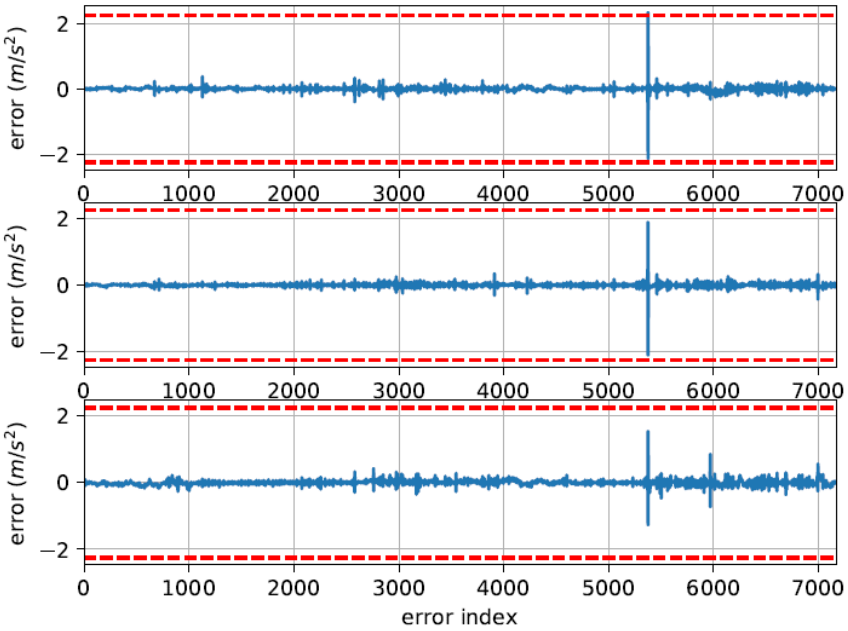

Figure 13 - acceleration error

Here it is noticeable that there was a single point where we received high error in acceleration that might be related to loosing track of calibration target as at that point of recorded dataset, we performed sharp movements as can be seen in force graph above.

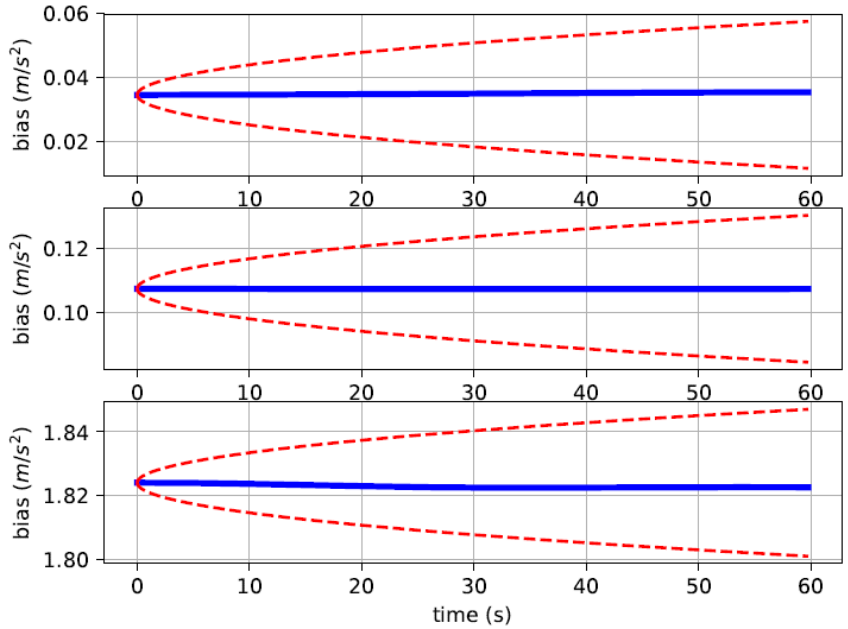

Figure 14 - estimated accelerometer bias (IMU frame)

Here we can see that we have estimated constant bias for each of the axes and it almost doesn't change over time which is a good sign for stability.

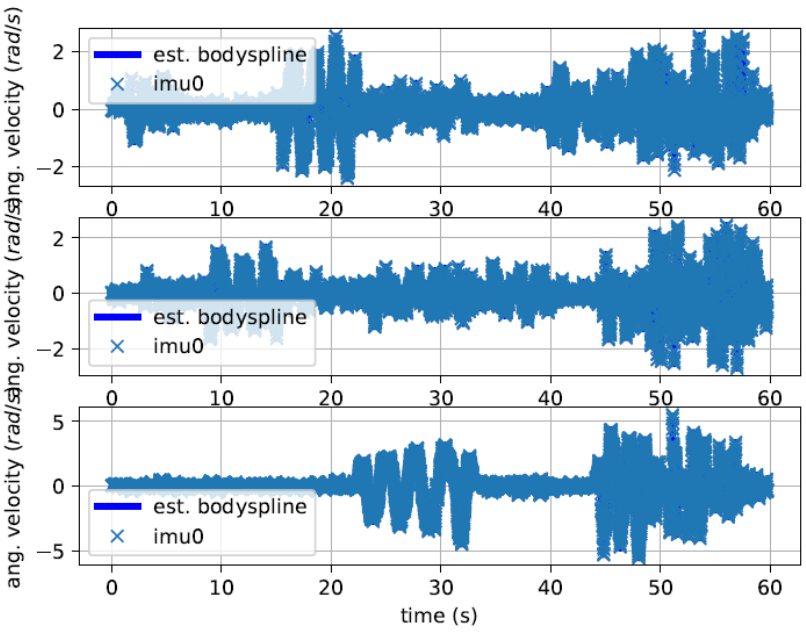

Figure 15 - comparison of predicted and measured angular velocity (body frame)

We can see in above figure that angular predicted and measured angular velocity corresponds to each other without visible out layers, moreover, change in movement at first half correspond to separated movements by different direction for pitch, roll and yaw.

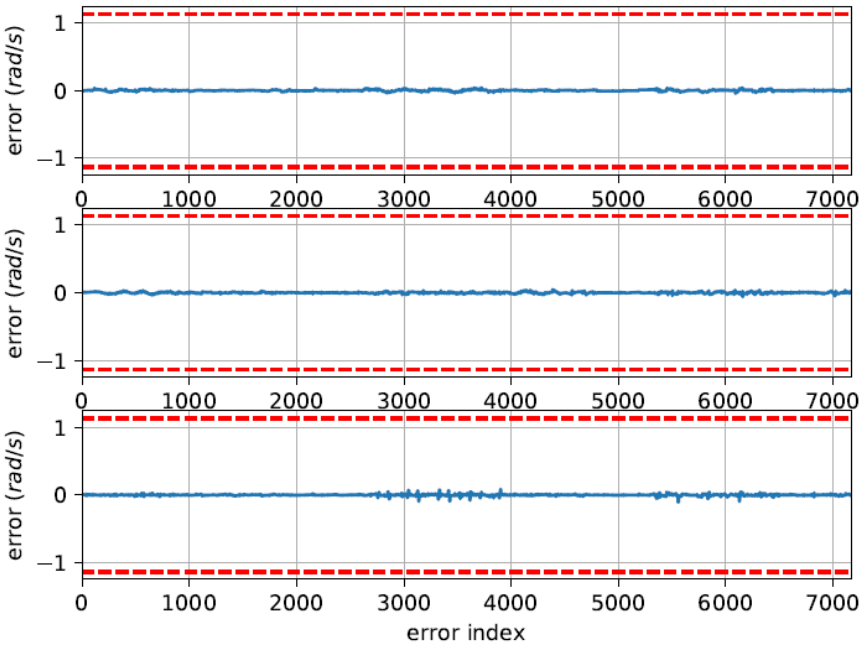

Figure 16 - angular velocity error

We can see that unlike the acceleration error, here we don't see any artifacts in behaviour.

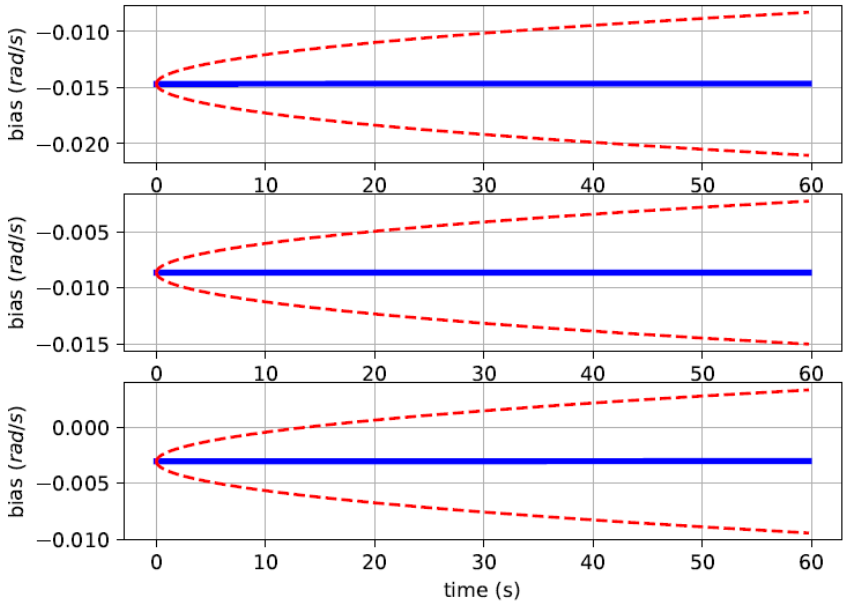

Figure 17 - estimated gyro bias (IMU frame)

Here we can notice that the estimated bias is constant too, like for accelerometer bias, that is a good result as there is no varying bias over time.

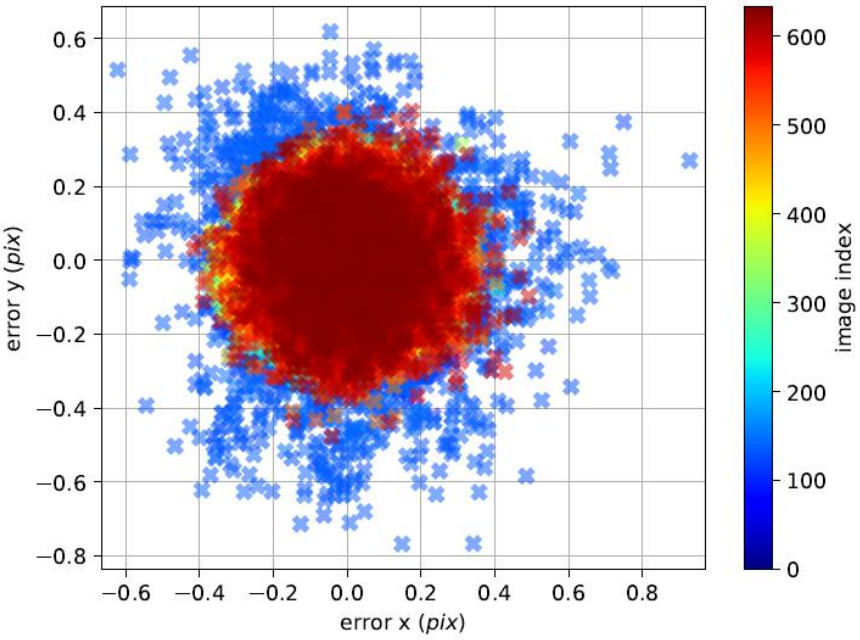

Figure 18 - reprojection errors

As it can be seen, the reprojection error stands in the desired range. It is below 1px error.

Now we will look at the theoretical rotation matrix versus calibration resulted and look at transformation vector measured versus acquired through calibration.

In our setup, we consider body position as the IMU position, hence we are looking to find transformation of camera position to IMU location.

By looking at the front view of the setup we can notice that the axes are as following:

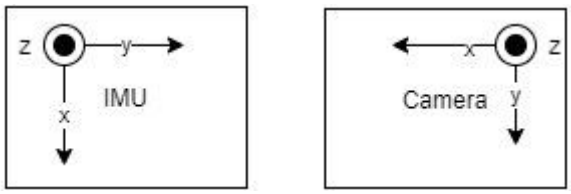

Figure 19 - IMU and camera theoretical axes

If so, rotation matrix will be:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{imu} = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{camera}$$

From calibration results we received following rotation matrix:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{imu} = \begin{bmatrix} -0.0072006 & 0.9999481 & -0.00720807 \\ -0.99973507 & -0.0070411 & 0.02191391 \\ 0.02186202 & 0.00736396 & 0.99973388 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{camera}$$

We received correct construction of rotation matrix.

Measured translation vector from camera to IMU with ruler in meters is:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{imu} = [0.015 \quad 0.03 \quad 0.002] \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{camera}$$

While we received:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}_{imu} = [0.01749895 \quad 0.02845419 \quad 0.01734108] \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{camera}$$

We can see that the numbers are verry close except for an error in Z axis, which we were not able to fix after several recalibration attempts.

Overall, it looks like the system was calibrated correctly.

7. Running the System

Prior to running the setup there is need to generate a configuration file that include values acquired by calibration process, example of configuration file that we generated for ORB-SLAM3:

```
%YAML:1.0

#--------------------------------------------------------------------------------------------
# Camera Parameters. Adjust them!
#--------------------------------------------------------------------------------------------
Camera.type: "KannalaBrandt8"
# Camera calibration and distortion parameters (OpenCV)
Camera.fx: 443.3541154029618
Camera.fy: 442.7629599414441
Camera.cx: 329.24635516913645
Camera.cy: 229.9852171160046
Camera.k1: 0.31825982073706915
Camera.k2: 0.2709370468036092
Camera.k3: -0.22108653885390658
```

```
Camera.k4: -0.1814796696573241

# Camera resolution
Camera.width: 640
Camera.height: 400

# Camera frames per second
Camera.fps: 20.0

# Color order of the images (0: BGR, 1: RGB. It is ignored if images are grayscale)
Camera.RGB: 0

# Transformation from body-frame (imu) to camera
Tbc: !!opencv-matrix
  rows: 4
  cols: 4
  dt: f
# Transformation from camera to body frame IMU
  data: [-0.0072006,   0.9999481,  -0.00720807,  0.01749895,
       -0.99973507, -0.0070411,   0.02191391,  0.02845419,
        0.02186202,  0.00736396,  0.99973388,  0.00734108,
       0.0, 0.0, 0.0, 1.0]

# IMU noise (Use those from VINS-mono)
IMU.NoiseGyro: 6.4424472128302748e-02 # rad/s^0.5
IMU.NoiseAcc:  1.2835782623650928 1e-01 # m/s^1.5
IMU.GyroWalk: 4.7533304166450236e-04 # rad/s^1.5
IMU.AccWalk:  1.8906248243325879 3e-03 # m/s^2.5
IMU.Frequency: 120

#--------------------------------------------------------------------------------------------
# ORB Parameters
#--------------------------------------------------------------------------------------------

# ORB Extractor: Number of features per image
ORBextractor.nFeatures: 1500 # Tested with 1250

# ORB Extractor: Scale factor between levels in the scale pyramid
ORBextractor.scaleFactor: 1.2

# ORB Extractor: Number of levels in the scale pyramid
ORBextractor.nLevels: 8

# ORB Extractor: Fast threshold
# Image is divided in a grid. At each cell FAST are extracted imposing a minimum response.
# Firstly we impose iniThFAST. If no corners are detected we impose a lower value minThFAST
# You can lower these values if your images have low contrast
# ORBextractor.iniThFAST: 20
# ORBextractor.minThFAST: 7
ORBextractor.iniThFAST: 20
ORBextractor.minThFAST: 7

#--------------------------------------------------------------------------------------------
```

```
# Viewer Parameters
#--------------------------------------------------------------------------------------
Viewer.KeyFrameSize: 0.05
Viewer.KeyFrameLineWidth: 1
Viewer.GraphLineWidth: 0.9
Viewer.PointSize: 2
Viewer.CameraSize: 0.08
Viewer.CameraLineWidth: 3
Viewer.ViewpointX: 0
Viewer.ViewpointY: -0.7
Viewer.ViewpointZ: -3.5
Viewer.ViewpointF: 500
```

The values to be adjusted are camera intrinsic parameters, fps, IMU parameters and extrinsic parameters.

Also, it is required to generate a roslaunch file which will be launched for the orb_slam wrapper that was written in the lab. Example of our launch file:

```
<launch>
  <!-- ARGS -->
  <arg name="camera_frame" default="front_camera_frame"/>
  <arg name="camera_optical_frame" default="camera_optical"/>
  <arg name="pose_topic" default="/orb_slam3_pose"/>
  <arg name="camera_topic" default="/camera/image_raw"/>
  <arg name="path_vocabulary" default="{PATH_TO_DIRECTORY}/ORB_SLAM3/Vocabulary/ORBvoc.txt"/>
  <arg name="path_settings" default="$(find orb_slam3_wrapper)/{PATH_TO_CONFIG_FILE}"/>
  <arg name="use_viewer" default="true"/>

  <arg name="orb_var_x" default="0.001"/>
  <arg name="orb_var_y" default="0.001"/>
  <arg name="orb_var_z" default="0.001"/>
  <arg name="orb_var_roll" default="0.001"/>
  <arg name="orb_var_pitch" default="0.001"/>
  <arg name="orb_var_yaw" default="0.03"/>

  <!-- ORB SLAM 3 -->
  <param name="camera_frame" value="$(arg camera_frame)"/>
  <param name="camera_optical_frame" value="$(arg camera_optical_frame)"/>
  <param name="pose_topic" value="$(arg pose_topic)"/>
  <param name="camera_topic" value="$(arg camera_topic)"/>
  <param name="path_vocabulary" value="$(arg path_vocabulary)"/>
  <param name="path_settings" value="$(arg path_settings)"/>
  <param name="use_viewer" value="$(arg use_viewer)"/>
  <param name="orb_var_x" value="$(arg orb_var_x)"/>
  <param name="orb_var_y" value="$(arg orb_var_y)"/>
  <param name="orb_var_z" value="$(arg orb_var_z)"/>
  <param name="orb_var_roll" value="$(arg orb_var_roll)"/>
  <param name="orb_var_pitch" value="$(arg orb_var_pitch)"/>
  <param name="orb_var_yaw" value="$(arg orb_var_yaw)"/>
  <node pkg="orb_slam3_wrapper" type="orb_slam3_mono_inertial" name="orb_slam3_mono_inertial"
output="screen"/>

</launch>
```

For VINS-MONO we have the following configuration file:

```
%YAML:1.0

#common parameters
imu_topic: "/imu"
image_topic: "/camera/image_raw"
output_path: "/home/vins/vins_output/"

#camera calibration
model_type: KANNALA_BRANDT
camera_name: camera
image_width: 640
image_height: 400
projection_parameters:
  k1: 0.31825982073706915
  k2: 0.2709370468036092
  k3: -0.22108653885390658
  k4: -0.1814796696573241
  mu: 443.3541154029618
  mv: 442.7629599414441
  u0: 329.24635516913645
  v0: 229.9852171160046

# Extrinsic parameter between IMU and Camera.
estimate_extrinsic: 0   # 0  Have an accurate extrinsic parameters. We will trust the following imu^R_cam,
imu^T_cam, don't change it.
                    # 1  Have an initial guess about extrinsic parameters. We will optimize around your initial guess.
                    # 2  Don't know anything about extrinsic parameters. You don't need to give R,T. We will try to
calibrate it. Do some rotation movement at beginning.
#If you choose 0 or 1, you should write down the following matrix.
#Rotation from camera frame to imu frame, imu^R_cam
extrinsicRotation: !!opencv-matrix
  rows: 3
  cols: 3
  dt: d
  data: [-0.0072006,   0.9999481,  -0.00720807,
       -0.99973507, -0.0070411,   0.02191391,
        0.02186202,  0.00736396,  0.99973388 ]
extrinsicTranslation: !!opencv-matrix
  rows: 3
  cols: 1
  dt: d
  data: [  0.01749895,  0.02845419, 0.01734108 ]

#feature traker paprameters
max_cnt: 150            # max feature number in feature tracking
min_dist: 30            # min distance between two features
freq: 10                # frequence (Hz) of publish tracking result. At least 10Hz for good estimation. If set 0, the
frequence will be same as raw image
F_threshold: 1.0        # ransac threshold (pixel)
show_track: 1           # publish tracking image as topic
```

```
equalize: 0            # if image is too dark or light, trun on equalize to find enough features
fisheye: 0             # if using fisheye, trun on it. A circle mask will be loaded to remove edge noisy points

#optimization parameters
max_solver_time: 0.04  # max solver itration time (ms), to guarantee real time
max_num_iterations: 8   # max solver itrations, 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: 1.2835782623650928 1e-01      # accelerometer measurement noise standard deviation.
gyr_n: 6.4424472128302748e-02        # gyroscope measurement noise standard deviation.
acc_w:  1.89062482433258793e-03         # accelerometer bias random work noise standard deviation.
gyr_w:  4.7533304166450236e-04        # gyroscope bias random work noise standard deviation.
g_norm: 9.81       # 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/vins/output/pose_graph/" # save and load path

#unsynchronization parameters
estimate_td: 1          # online estimate time offset between camera and imu
td: 0.0138                # initial value of time offset. unit: s. readed 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             # unit: s. rolling shutter read out time per frame (from data sheet).

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

and following launch file:

```
<launch>

   <arg name="config_path" default = "$(find feature_tracker)/../config/arducam/arducam.yaml" />
   <arg name="vins_path" default = "$(find feature_tracker)/../config/../" />

   <node name="feature_tracker" pkg="feature_tracker" type="feature_tracker" output="log">
      <param name="config_file" type="string" value="$(arg config_path)" />
      <param name="vins_folder" type="string" value="$(arg vins_path)" />
   </node>

   <node name="vins_estimator" pkg="vins_estimator" type="vins_estimator" output="screen">
      <param name="config_file" type="string" value="$(arg config_path)" />
      <param name="vins_folder" type="string" value="$(arg vins_path)" />
   </node>

   <node name="pose_graph" pkg="pose_graph" type="pose_graph" output="screen">
```

```
    <param name="config_file" type="string" value="$(arg config_path)" />
    <param name="visualization_shift_x" type="int" value="0" />
    <param name="visualization_shift_y" type="int" value="0" />
    <param name="skip_cnt" type="int" value="0" />
    <param name="skip_dis" type="double" value="0.1" />
  </node>

</launch>
```

We provide a series of commands that should be executed with filled in information depending on ones need. To use the setup without creating automation scripts, there is need to run commands on RPi and on host PC.

First, configure host PC to be ROS master by setting environment variable as bellow, it will be the roscore instance to which RPi will publish the topics:

Run on host:

```
export ROS_MASTER_URI=http://localhost:11311
roscore
```

Afterwards log in to RPi through SSH - either from 2 separate sessions then run:

```
export ROS_IP={fill in IP of RPi}
export ROS_MASTER_URI=http://{fill in IP of HOST pc}:11311
sudo pigpiod - (Or set is startup service)
source ~/{folder_where_mpu9150_installed}/devel/setup.bash
rosrun mpu9150_imu_pi imupi_python.py
```

from another SSH session - run:

```
export ROS_IP={fill in IP of RPi}
export ROS_MASTER_URI=http://{fill in IP of HOST pc}:11311
source ~/{folder_where_arducam_stream_installed}/devel/setup.bash
roslaunch cpp_stream cam_stream.launch gain:={desired_gain} exposure:={desired_exposure}
```

On host PC open another 3 (4 for VINS-MONO) terminals:

in terminal A - run:

```
rosrun image_transport republish compressed in:=/camera/image raw
out:=/camera_decompressed/image_raw
```

in terminal B - run:

```
source binning_ws/devel/setup.bash
rosrun binning binning_node
```

in terminal C – run for ORB-SLAM3:

```
source ./lab3/lab3_ws/devel/setup.bash
roslaunch orb_slam3_wrapper orb_slam3_mono_inertial.launch
```

or

```
source ./vins_mono_ws/devel/setup.bash
roslaunch vins_estimator arducam.launch
```

for VINS-MONO open another terminal – D and run RVIZ configuration:

source ./vins_mono_ws/devel/setup.bash

roslaunch vins_estimator vins_rviz.launch

If all was correctly configured, the orb_slam3 node should start and 2 windows should pop up, one window will show video stream with feature squares on it, the other window will show the generated map, local features and marked key frames.

Example result for running the setup:

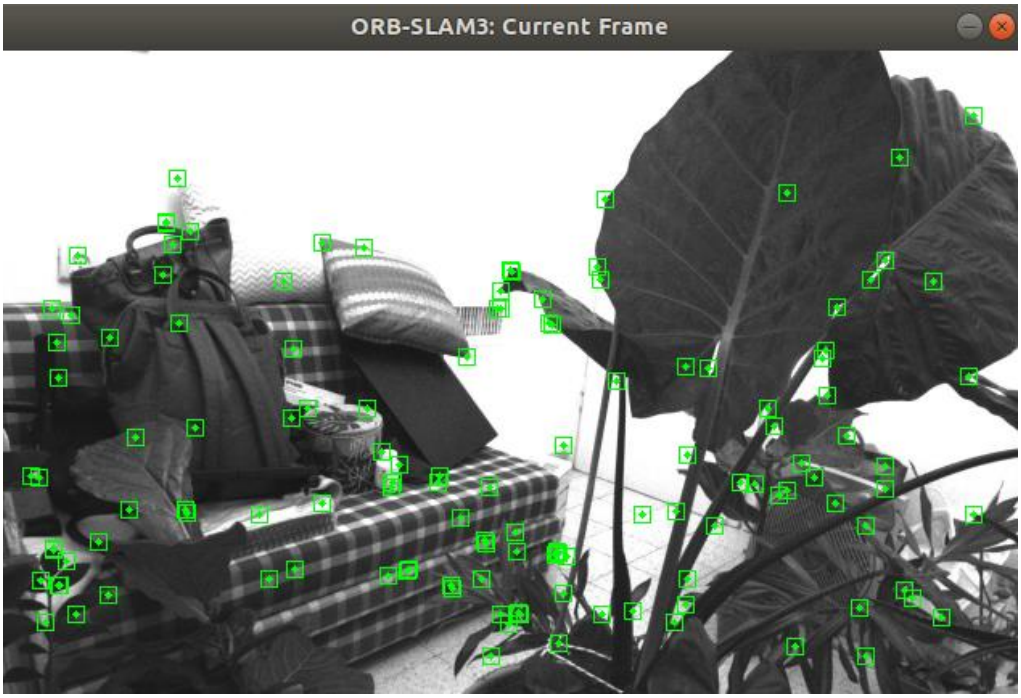

Figure 20 - current frame with features ORB SLAM3

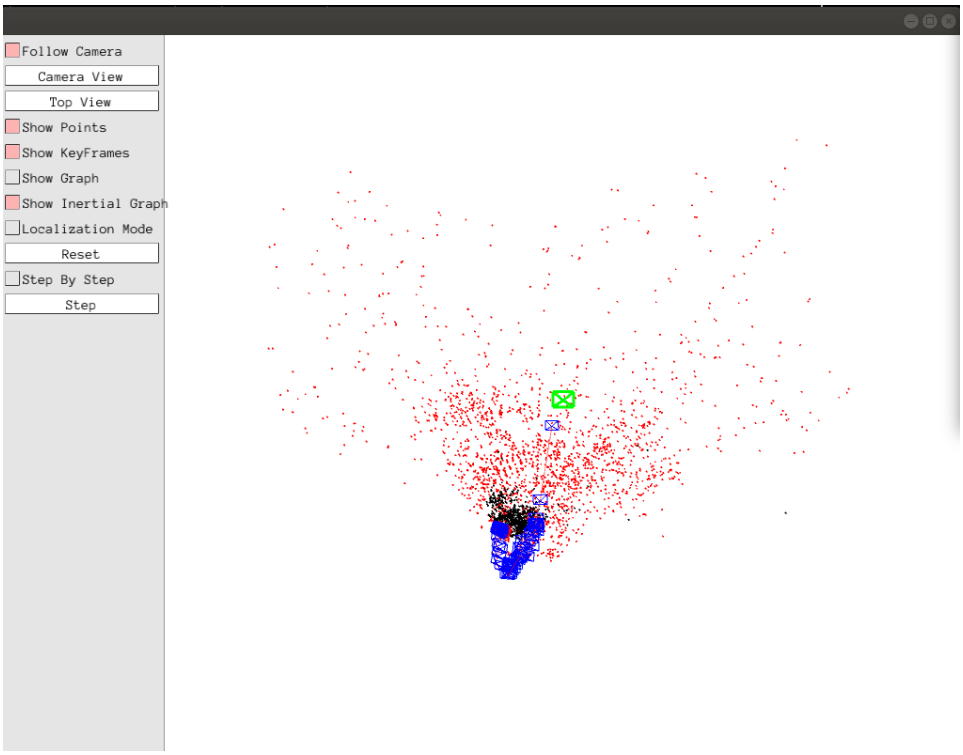

Figure 21 - current map with local features and key frames

For VINS-MONO we should receive a RVIZ window which outputs current frame with, estimated map and window showing loop closure when revisiting close frames again:

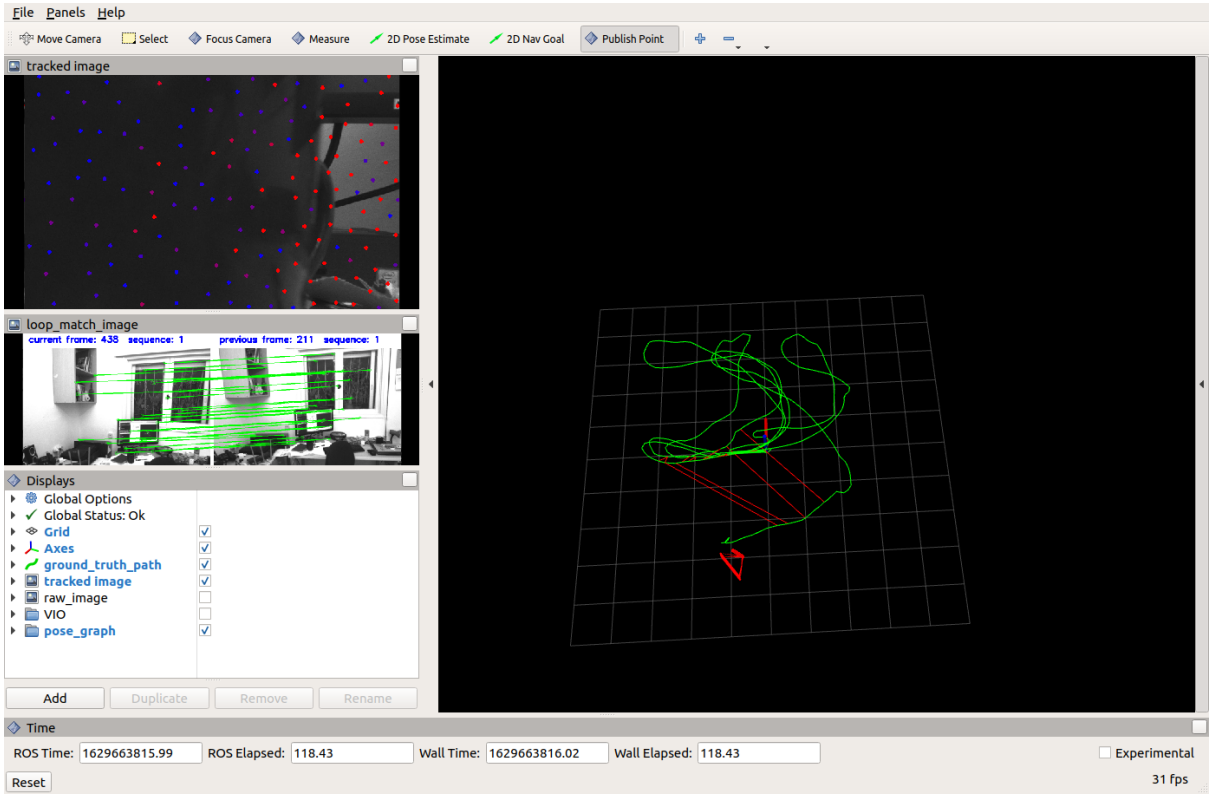

Figure 22 - VINS-MONO tracked image, map, and loop match image

8. Results

Prior to running the algorithm on our setup, a test scenario from EuRoC MAV dataset was used, both for ORB-SLAM3 and VINS-MONO. We run dataset – Machine Hall 01 – all datasets can be found on https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets.

By running the dataset, we could observe that first few seconds before drone take off, it was period of initialization which made the algorithm lock in and generate initialized map. After that the drone took off and flew smoothly around the room which was rich in texture and spacious. It could be observed that by flying around the room, ORB-SLAM3 generated continues map with key frames on them generated at 4Hz. We could observe how the algorithm fixes the track and performs bundle adjustment to fix map misalignments. At the end of dataset algorithm performs good loop closure as drone returns to initial point.

With VINS-MONO we can observe similar behaviour with a small difference, the calibration sequency is shorter. It can be noticed that feature tracker result in more scattered features across the frame. Besides this, and some data representation differences on tracked images and map there was no noticeable difference.

When we ran ORB-SLAM3 in online mode with our setup, we encountered several issues with initiating new map, as according to the paper of ORB-SLAM3, first few seconds are initiation period where algorithm collects data for IMU calibration for proper scale factor calculation.

We could not obtain good results in starting the algorithm in inertial mode, only several times it managed to lock in, and afterwards it possessed good results in following and generating a map with good loop closure after returning to previous visited key frames.

For comparison we checked performance of pure visual SLAM instance which acquired good tracking results but without correct scale factor, as it was expected for mono camera case. We can assume that data combination of IMU and camera image results in some type of mismatch. Going over discussions in the repository it is pointed as a possible main issue for bad results.

We also applied VINS-MONO algorithm on our setup which succeed in tracking but had a drift in revisiting previous tracks when we performed movement for several meters, as could be seen in figure 22. Bundle adjustment didn't fix the drift. But on the other hand, we received a map unlike in ORB-SLAM3 where it failed to lock in.

We also, used VINS-MONO to generate a rotation and translation matrix as a sanity check. Values it generated were less accurate than those generated using Kalibr with error of several centimetres.

We eliminated possibility of time tags mismatch in messages, as they weren't changed during process and were generated as close to data generation as possible. A constant mean delay fix was inserted into the ORBSLAM3 wrapper to fix the constant delay, the value is given by Kalibr and stated around 15ms delay between IMU and camera. Yet we did find out that at higher rates of IMU node publisher is having drift in rate of update, which might be a cause of error if the algorithms relay strongly on time tags when working with several samples between 2 corresponding frames.

We didn't create auto exposure and gain control which caused saturation of light in the sensor when moved to bright environment from dark one, and vice-versa. The algorithm had hard time on tracking features in such scenarios.

9. Discussion and Conclusion

We didn't achive results we hoped for, and therefore we were unable to perform good comparison between running the system with inertial sensor and without to show improvement in resulted tracked map.

Nevertheless, it can be observed that there was done a pioneering work in the field when running the dataset on the algorithm.

Moreover, we made a proof of concept that a low cost, low power system can generate synchronised constant stream of data, IMU and image with good resolution (originally, we published 1280x800 resolution images which was squashed on host side) with a small delay (was not measured) proportion to real time. This delay can be observed by looking at the presented image stream. This delay can be estimated of few milliseconds.

Also, we showed that it is possible to stream data online through limited bandwidth with high throughput without loosing information.

Future work needs to be done on this setup as we were not able to find the cause of failure of algorithm work or inconsistent bundle adjustment when revisiting same frames.

It might be due software failure to generate consistent node operation rate for IMU python topic, due to python being script and not compiled language and it has higher overhead for embedded systems then for high processing systems.

10.Future work

As the results achieved were not satisfying, the ORB-SLAM3 didn't manage to work fluently with our setup, several investigation topics and improvements can be suggested.

Firstly, we didn't implement auto exposure and gain balance algorithm as for this time of writing it was now natively supported by Arducam driver for external trigger mode this might improve algorithm adaption to light environment change and improve feature extraction.

Secondly, we didn't measure time delays from camera request for image and image receive by Arducam, even thou calibration process achieved good results.

Thirdly, we didn't investigate algorithm work with recorded data from our setup to find fail points.

Fourthly, we didn't try other lenses with greater FOV which can improve features extraction in indoor small environments like the setting we checked in. As it stated in the paper their test environment was above 5m distance while ours below.

Fifthly, we didn't manage to acquire correct distance in Z axis in calibration and therefore we can't ignore this fact potentially affecting the algorithm work.

Sixthly, the researchers that developed the code released new beta improved version that wasn't checked.

Seventhly, a hardware generated trigger should be work better than software operation rate generator, which should create much more consistent performance and higher rates of operation.

Lastly, it might be a good idea to check performance of our setup on other versions of ORB-SLAM algorithms.

11. References

[1] - https://github.com/UZ-SLAMLab/ORB_SLAM3 – official git repository of ORB SLAM3

[2] - https://arxiv.org/pdf/2007.11898.pdf - ORB SLAM3 paper

[3] - https://github.com/ArduCAM/MIPI_Camera - Arducam git repository and wiki

[4] - https://www.raspberrypi.org/software - official site of RapberryPi

[5] - https://github.com/ethz-asl/kalibr - git repository of Kalibr project

[6] - https://github.com/gaowenliang/imu_utils - git repository of IMU utilities project

[7] - https://projects.asl.ethz.ch/datasets/doku.php?id=kmavvisualinertialdatasets – EuRoC dataset site

[8] - https://github.com/HKUST-Aerial-Robotics/VINS-Mono - VINS-Mono git repository

[9] - https://github.com/tau-adl/mpu9150 - git repository form TAU-ADL repository, of Python MPU ROS node

[10] - http://wiki.ros.org/ROSberryPi/Installing%20ROS%20Melodic%20on%20the%20Raspberry%20Pi – ROS melodic installation wiki for RPi

[11] - http://wiki.ros.org/melodic/Installation/Ubuntu - ROS melodic installation wiki for Ubuntu

[12] - https://github.com/tau-adl/Arducam_RPi_ROS_node - git repository of our Arducam ROS node

[13] - https://github.com/tau-adl/image_binning_ros_node - git repository of our binning ROS node

[14] - https://github.com/HKUST-Aerial-Robotics/VINS-Mono - official git repository of VINS-MONO

[15] - https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8421746 – VINS MONO paper