TEL AVIV UNIVERSITY
School of Electrical Engineering

Parallel Tracking and Mapping Algorithm Analysis For Drone Applications

A project submitted toward the degree of
Master of Science in Electrical and Electronic Engineering

by

Omer Segal

This research was carried out in The School of Electrical Engineering
Under the supervision of Mr. Yonatan Mandel and Prof. Ben-Zion Bobrovsky

June 2019

Contents

Abstract **2**

1. Introduction **3**

2. PTAM Implementation Analysis **5**
2.1 PTAM Wrapper 5
2.2 Tracking 8
2.3 Mapping 14
2.4 Flow Charts 15

3. Background Terms **19**

4. PTAM Configuration **23**

5. Installation **29**

6. Debug **30**

7. Discussion **31**

8. Conclusion **32**

9. Bibliography **33**

Abstract

In this work, we have provided a set of tools for adopting a known implementation of a computer vision based simultaneous localization and mapping (SLAM) algorithm for pose estimation.
These tools are meant to enable to fully understand the given implementation, its assumptions and abilities, and to understand how to modify it for a future use.

First, we created a detailed documentation of the algorithms implementation along with flow charts to clarify the high level of the code.
Second, we gave a theoretical review that links the implementation to relevant computer vision terms for a better understanding of each step in the algorithm.
Third, we gathered a guide for installing and debugging the code environment for creating the ideal workspace.

Alongside these tools, we added relevant recommendations and insights that can be useful for modifying the implementation, as well as figures from our experiments that are given as visualization aids.

1 Introduction

In navigation, robotic mapping and odometry for virtual reality or augmented reality, simultaneous localization and mapping (SLAM) is the computational problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent's location within it. SLAM algorithms are tailored to the available resources, hence not aimed at perfection, but at operational compliance. Published approaches are employed in self-driving cars, unmanned aerial vehicles, autonomous underwater vehicles, planetary rovers, newer domestic robots and even inside the human body.
There are many algorithms known for solving it, at least approximately, in tractable time for certain environments. One of these algorithms is the parallel tracking and mapping (PTAM) algorithm that was introduced by G. Klein and D. Murray et al. in 2007 [1].

The PTAM algorithm was designed for small workspaces and suggested to split tracking and mapping into two separate tasks, processed in parallel threads - one thread deals with the task of robustly tracking erratic motion, while the other produces a 3D map of point features from previously observed video frames. This allowed the use of computationally expensive batch optimization techniques not usually associated with real-time operation.

PTAM is also a feature-based SLAM algorithm that tracks and maps many (hundreds of) features to achieve robustness. Simultaneously, it runs in real-time by creatively parallelizing the motion estimation and mapping tasks and by relying on efficient keyframe-based Bundle Adjustment (BA) instead of Bayesian filtering for pose and map refinement, which are two main reasons to make PTAM outperform MonoSLAM and the like in both efficiency and precision.

Other SLAM algorithms tend to focus on improving specific components, while the high level logic remains similar. For example, the usage of improved image feature detector and descriptors like ORB [2] provide real-time performance with desktop CPU which SIFT [3] or SURF [4] cannot provide, while the usage of some nonlinear least square optimization libraries like g2o [5] provide utilities to do bundle adjustment, etc. Among them ORB-SLAM [6] by R. Mur-Artal et al. and LSD-SLAM (Large-Scale Direct Monocular SLAM) [7] by J. Engel et al. received much appreciation from the community.

As we have mentioned, the PTAM algorithm was a main turning point in the usage and performance of SLAM, and is used as a base structure to many other algorithms.
For that reason, we have chosen to focus this work on the PTAM algorithm, and specifically on its implementation in J. Engel's paper - “Autonomous Camera-Based Navigation of a Quadrocopter” [8].

While the implementation follows the guidelines of G. Klein and D. Murray paper, it combined numerous steps that were essential in order to implement the algorithm on a drone data stream.
Adding these steps required a deep understanding of the PTAM logic and many computer vision and photogrammetry methods and tools.

Our work provides tools that are meant to enable to fully understand the given implementation, its assumptions and abilities, and to understand how to modify it for a future use, without the necessity of vast prior knowledge.

Chapter 2 will review and analyze the given PTAM implementation in a step by step manner along with a flow chart that will be referred to for a better grasp of the algorithm.
In Chapter 3 we will overview basic computer vision terms that are essential for understanding the PTAM flow.
In Chapter 4 we will go through the configuration options for the PTAM implementation and provide insights and intuition for each modification.
Chapter 5 will list the steps for installing the code packages and creating a suitable debug environment.
Chapter 6 will provide instruction for an easy way to debug the code and the PTAM process.

2 PTAM Implementation Analysis

In this chapter, we will review the PTAM implementation, explain the logic behind it and analyze its assumptions and parameters.
During this chapter we will refer to relevant background terms that are elaborated in the next chapter, by adding [T#] next to the term.
In addition, [G#] next to a method will point a reference to the flow chart graph that provides a graphic understanding of the code structure.

The script that allows us to focus on the state estimation part of the code and enables us to understand and debug specifically the PTAM flow is called "main_stateestimation".
In it we can find the estimation node that contains two important functionalities – the ptamWrapper and the mapViewer.
The mapViewer displays the map that is being built during the PTAM flow and is used for debug purposes.
The ptamWrapper is entity that handles the video and navigation data stream and applying the PTAM flow – meaning, tracking (by the Tracker) and mapping (by the MapMaker).
From now on, we will focus on the ptamWrapper and the code flow it carries.

2.1 PTAM Wrapper

The ptamWrapper is implemented as a thread with the basic run function – PTAMWrapper::run() [G1].
This run function contains two parts:

1. An initiation method that happens once - PTAMWrapper::ResetInternal().
2. A per frame logic - PTAMWrapper::HandleFrame().

Let us review these two parts.

PTAMWrapper::ResetInternal() [G2]:
As we have mentioned, this part of the run function is called once, at the beginning of the flow in order to initiate the code environment.
The initiation includes:

1. Reading the camera calibration and initiating the camera object with the calibration parameters: x focal length, y focal length, x offset, y offset and a distortion parameter (w).
2. Initiating the map object to include later on MapPoints and KeyFrames.
3. Initiating the MapMaker thread, that is in charge of the mapping part of the PTAM.
4. Initiating the Tracker, that is in charge of the tracking part of the PTAM. The Tracker is initiated with the image size, the camera object, the Map object and the MapMaker thread.

Figure 1 demonstrates the main components of the PTAM – MapPoints and KeyFrames.
The KeyFrames are the small 5 image planes positioned in different locations and orientations.
The MapPoints are the 3D points cloud that creates the map (the house, in the figure) after the bundle adjustment process.

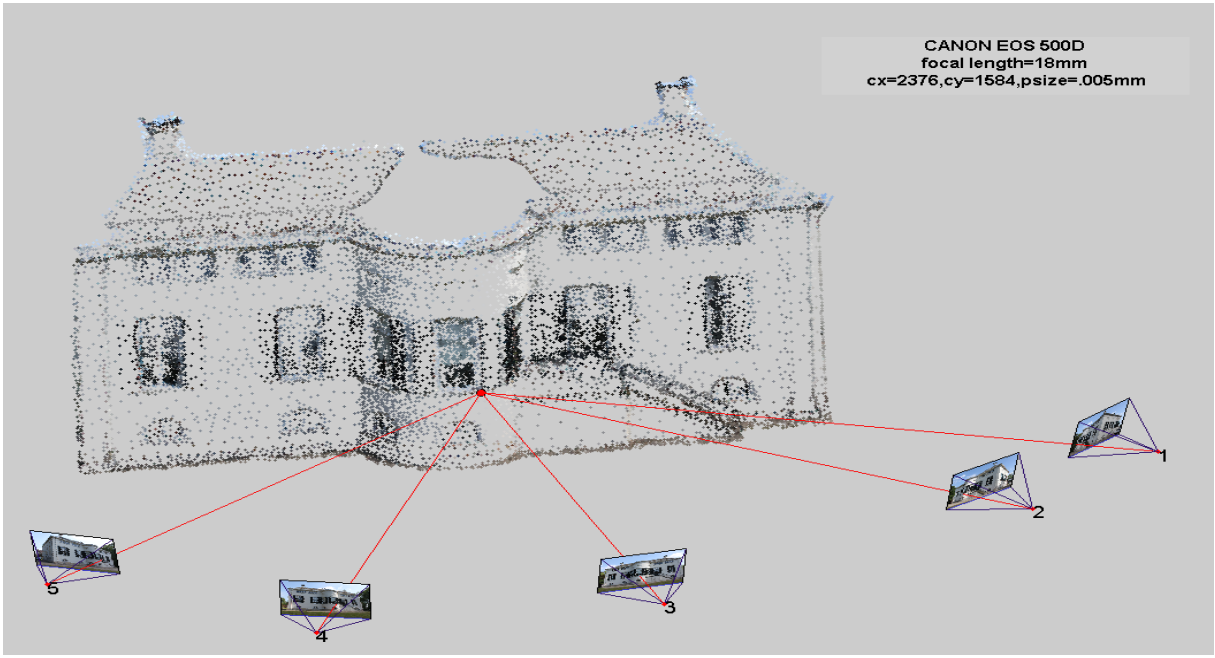


Figure 1: Multi image intersection by interactive point measurements using bundle adjustment (www.mathworks.com)

PTAMWrapper::HandleFrame() [G3]:

As we have mentioned above, this is the main functionality, which is called for every new frame. For every new frame, the HandleFrame() method:

1. Estimates an initial guess for the camera pose
2. Performs the tracking logic and improves the pose estimation
3. Decides whether this frame is a suitable candidate for a keyFrame or not
4. Adds the keyFrame to the mapMaker queue (if that is the decision in step 3)

Let us elaborate on each step we have mentioned.

1. Initial pose estimation [G4]:
   For each frame, we can express the camera pose with 6 parameters – 3 for location (X, Y and Z coordinates) and 3 for orientation (roll, pitch and yaw angles). This can be referred as the frame camera model.
   We want our initial pose estimation to be as good as possible in order to narrow for the Tracker search range, so it will perform better (we will explain in it extensively later on).
   In order to have a first guess for the transformation from the previous frame camera model to the current frame camera model, we use the estimation of the Kalman filter [T1] as follow:
   a. Get the current drone pose estimation using the Kalman filter – by using the getPoseAtAsVec() method.
   b. Get the previous pose estimation by subtracting the last Kalman filter offset, as XYZ coordinates and roll, pitch and yaw angles – by using the backTransformPTAMObservation () method.
   c. Translate the orientation angles into a 3X3 rotation matrix and calculate the transformation to the front camera and the bottom camera. This is done by using the Predictor::setPosRPY() method. Note that the transformation offsets are hard coded in Predictor::droneToBottom and in Predictor::droneToFront.
   d. Translate the drone pose into the front camera pose, since we are interested in the camera model.

2. Tracking logic [G5]:
   Will be elaborated in the Tracking section.

3. Tracking result evaluation [G6]:
   At this point we can assess if the tracking result is reliable by observing two conditions:
   a. We compare the prediction of the Kalman filter to the pose given by the PTAM tracking result - If the difference in the angles is too big, then the result is marked as bad (note that the angles thresholds are hard coded). Be advised that if we want to create a scenario where the Kalman filter is disabled or not reliable, this condition must be adjusted.
   b. In case the first condition is met, we need to check whether the last set of tracking results for the last set of frames was good, since a single tracking result is not strong enough. For every tracking result, we update the length of the strike of good results, so it could be observed - If the good results strike is long enough, then we can consider taking this frame as key frame. Note that the threshold for a long enough strike is hard coded to 10 frames. Be advised that in case of a scenario where the bad tracking frames are more common, this threshold could be lowered.

4. KeyFrame insertion [G7]:
   This step is done by the Tracker::TakeKF() method.
   If the tracking quality is good and a new key frame is needed – we add the current frame as a new key frame by adding it to the MapMaker's queue. We do it by calling the Tracker::AddNewKeyFrame() method.
   We check if a new key frame is needed by observing two conditions:
   a. Minimum required time between the current frame and previous keyFrame that was added.
   b. Minimum required distance between the current frame and the closest keyFrame that already exists in the map.
   The explanation for modifying these conditions is given the configuration part.

Figure 2 provides a visualization of the PTAM result. In the figure we can see the MapPoints that appear in the frame and the assessment of the result quality. In the bottom of the figure, we can see two data lines: The first one contains the general that status of the algorithm (in our case – "Tracking Map"), the status of the tracking quality (in our case – "good"), the number of map points that were found in each scale (will be elaborated in the section 2.2) and the number of all the map points that the map contains. The second line provides information on the overall performance of the PTAM process and the map scale.

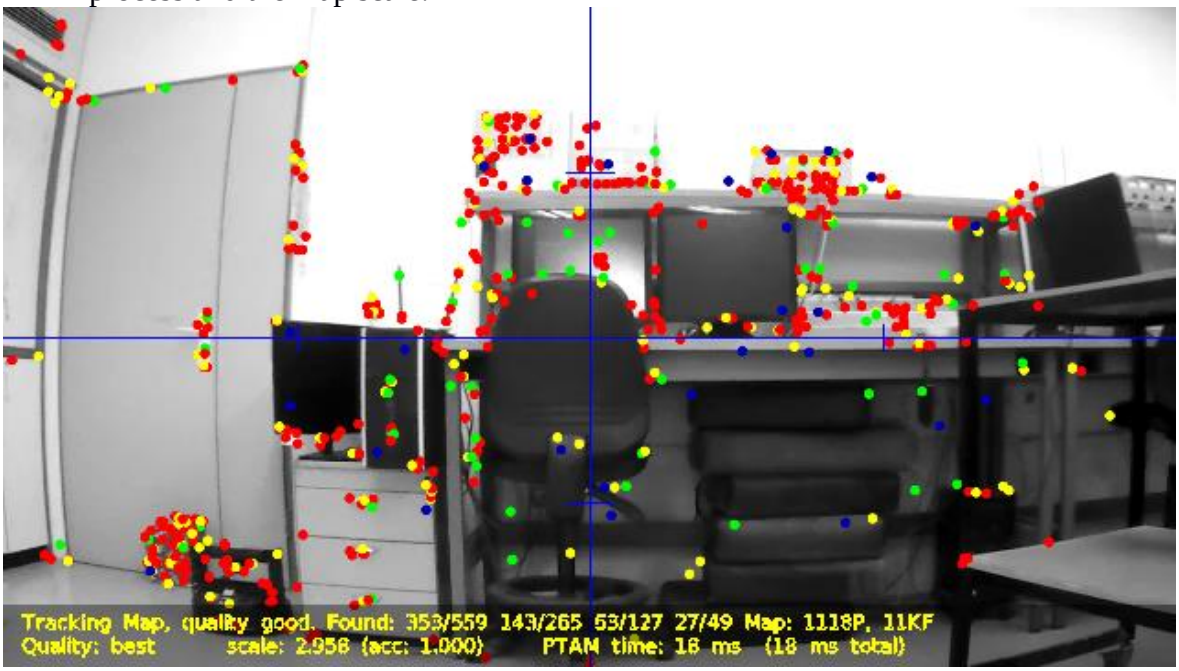


Figure 2: PTAM Result visualization on the video stream

2.2 Tracking

This part of the PTAM is performed by the Tracker::TrackFrame() method.
The Tracking stage includes a preparation step and then splits into two logics that depend on whether a map already exists at this point or not.

Tracking preparation steps [G9]
This is done by calling the KeyFrame::MakeKeyFrame_Lite() method.
This method prepares the frame object from an image:

1. Generate a Gaussian pyramid from the image –
   Each level in the pyramid is a blur and down sample of the previous level, where the starting level is a full scale of the image. The number of levels in the pyramid is defined by LEVELS (an explanation for modifying it is given the configuration part). Generating the pyramid allows us to control the level of accuracy we look for at any point of the algorithm. Meaning, when working on the zero level of the pyramid (full scale) we get the maximum precision. While, when working on a higher level, we will achieve better processing time performance on the expense of accuracy. The blur part is highly important, since without it the down sampling will cause losing a lot of information when creating the next pyramid level.

2. Detect and store FAST corner points –
   For each level in the Gaussian pyramid, we want to detect points of interest in the image (in the relevant scale). These points of interest are feature points that will be tracked later on and could also become map points. There are many ways to detect feature points in an image, and here it is done using the FAST corner detection algorithm [T2]. Note that for each level we use a different threshold for the detection algorithm.

The last preparation step is to keep a blur of the current frame that will be used later on for improving the frame pose guess (elaborated in the second tracking branch).

First Tracking branch – No map was created [green branch]
If we don't have a map at this point of the flow, need to establish the initial map – by calling the Tracker::TrackForInitialMap() method [G11].
This part requires two spacebar presses from the user in order to define the first two key-frames, as we will explain.

When the user presses the spacebar for the first time, we set the current frame as the first key frame by calling the Tracker::TrailTracking_Start() method [G13].
The main functionality in this method is performed by KeyFrame::MakeKeyFrame_Rest() -
For each level in the pyramid, we find FAST corners which are maximal (by performing a non-maximal suppression [T3] on this set of FAST features).
Afterwards, we calculate the Shi-Tomasi scores [T4] of those corners, and keep the ones with a suitably high score as candidates, i.e. points which the mapMaker will attempt to make new map points out of. Note that we keep only the points from the high-resolution level of the Gaussian pyramid.

Once the first keyFrame was set, we start building trails for the corners in it to the matching corners in the current frame, by using the Tracker::TrailTracking_Advance() method [G14].
We Go over all the points in the previous frame and compare it as a patch to all the corners in the current frame. The patch is a window of pixels around the corner point and the comparison is performed using sum of squared difference (SSD).

The corner is compared only if it is within the allowed range from the point in the previous frame, meaning 10 pixel difference in each axis. This implies that the initiation process can handle only slow movement this way.
Then, we perform the same logic only backwards – we search for the found corner in the previous frame. Only if we match these two patches both forward and backwards ("marriage matching"), we can be sure that the matching is strong and reliable.
Each point in the first keyFrame that lasts this tracking is represented as a trail from an original corner in the first keyFrame and to a corner in the current frame.
If we don't have enough trails that last [G15] – we reset the initiation stage [G16]. This threshold in hard coded to 10 trails.

Figure 3 provides a visualization of the above trails – from each feature point in the first keyFrame a trail in drawn to the pixel of the feature point in the current frame.

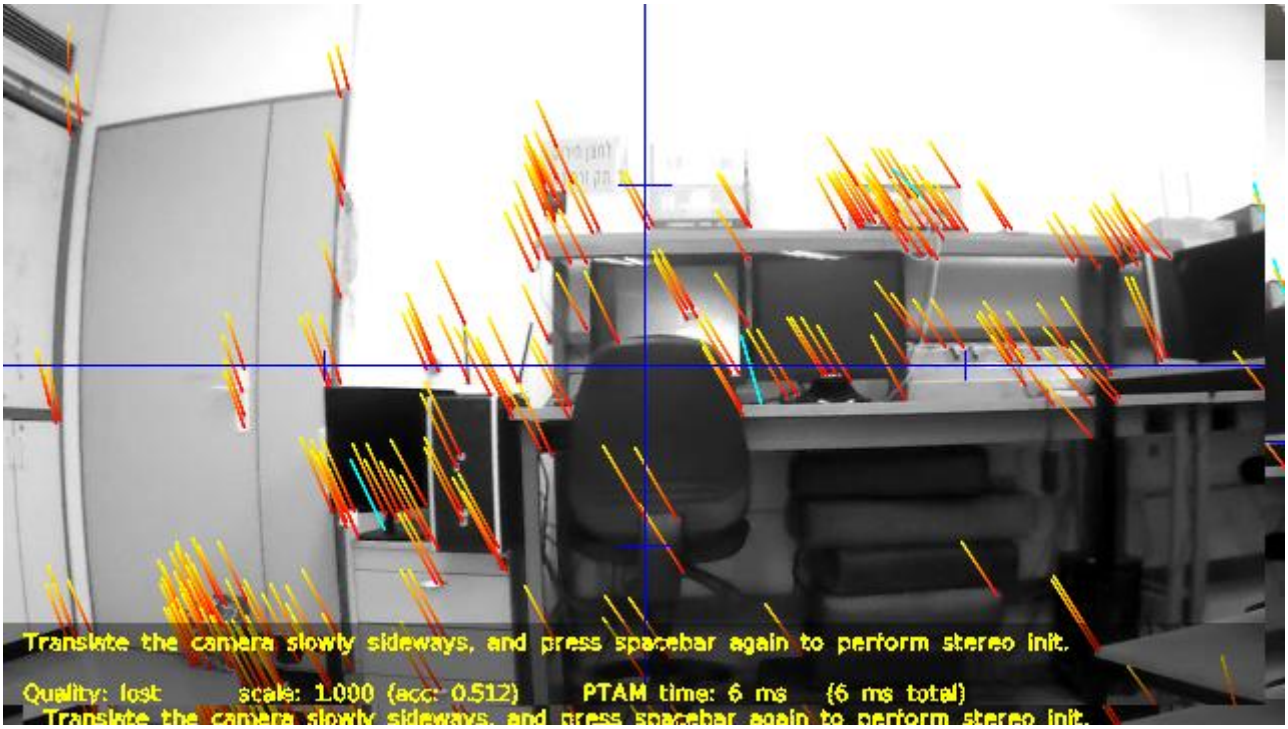


Figure 3: Trails visualization while initializing the map

This process continues until we reach the second keyFrame the user mark (by pressing the spacebar for the second time [G17]). When we do, we initiate the map using the trails that were found - by calling the MapMaker::InitFromStereo() method [G19].

1. We try to find a homography [T5] using the matches we have found (HomographyInit::Compute() method [G20]):
   a. Find the best homography using RANSAC [T6]
   b. Refine it by using only the inliers from the previous step
   c. Decompose the best homography into a set of possible decompositions of 3D translation and rotation (between the two frame camera models)
   d. Choose the best decomposition based on visibility constraints –
      i. Both frames must be in the same side of the object plane.
      ii. For all the reference points being visible, they must be in front of the camera.

   This step is performed by the method ChooseBestDecomposition() [9].

2. We check that the baseline between the two frame camera models is sufficient [G21] - The homography found must contain a translation part and the IMU [T7] must indicate a baseline larger than 5 cm.

3. Based on the decomposition that was found, we perform for every matching point a sub-pixel patch matching, followed by a triangulation [T8]. Then, the point is inserted into the map DB [G22] along with the two measurements from the first and second keyFrames. The insertion into the DB will be elaborated in the mapping section.

4. We perform 5 iterations of Bundle Adjustment (BA) [T9], on the initialized map with the two keyFrames, while the first keyFrame is fixed (meaning, its model parameters remain unchanged) and the second isn't.

5. Now, we need to expand our map to include more points (still from the two key frames). In order to do so, an epipolar search [T10] is needed, which requires a new calculation of the scene depth from each keyFrame. We run MapMaker::AddSomeMapPoints() [G23] for every level in the pyramid. The main method in it is the epipolar search function for every corner of the first frame in the relevant pyramid level - AddPointEpipolar() [G24]:
   a. Find the relevant epipolar line in the second frame
   b. Check all the corners in the second frame that are on this line by doing a zero-mean SSD with the given point from the first keyFrame.
   c. Take the best match and insert it to the map after triangulation on the found matching points.

6. At this point, our map includes many points and we can perform a BA again until full convergence [G25].

7. After we have updated the map, we normalize its scale to fit the depth scene from the first keyFrame, and then rotate and translate the map [G26] so the dominant plane is at z=0. Note that the map transformation is only a convention and isn't mandatory.

A second look at Figure 2 will show that the MapPoints are divided into 4 colors – each color stand for a different level of the pyramid which this point belongs to.

Second Tracking branch – A map was already created (orange branch)

If a map already exists and we are not lost, we want to improve our camera model guess:
The orientation part [G29]–
we calculate the 2d transformation from the last key frame by ESM-tracking a la Benhimane & Malis [T11] in an iterated process (Tracker::CalcSBIRotation() method). Note that the max iterations number is hard coded to 6 iterations.
Now we can use the 2d transformation we have calculated in order to improve our pose guess, specifically we look to improve the orientation part. We do it by solving only the rotation parameters, since we cannot solve the translation part without a given scale to the problem.
We decide if we are lost by observing the number of frames that led to bad tracking result quality from the last keyFrame that was added. Note that the threshold for the number of lost frames is hard coded to 3 frames.
The translation part [G30]–
For this, the Tracker keeps a motion model that is kept as a decaying velocity model.
The new pose is estimated by adding the velocity over 1 second to the previous pose (Tracker::ApplyMotionModel() method). As we can understand, this motion model assumes a one

frame per second frame rate, but it has no importance since we don't really have the actual scale of the scene (only a relative one).

Now that we have a roughly improved estimation of the current frame model, we can improve it even better by solving it with respect to the map we have created so far – by applying the main purpose of the Tracker, the Tracker::TrackMap() method [G31].

1. Since we have an estimation of the current frame model, we can use it in order to reduce the map to only potentially-visible-set (PVS) map points [G32]. Meaning, we project all the map points, and keep only the ones that are in the current frame model field of view.

2. After we have reduced the map points to PVS only, we want to reduce our search for each point to the relevant scale, so we will look for a match only in the relevant pyramid level of the frame. In order to do so, we run PatchFinder::CalcSearchLevelAndWarpMatrix() [G33] for each PVS point. This method finds the warping matrix and search level given the MapPoint, the current frame camera model and the projection derivatives (meaning, the derivatives of the projection in the previous frame and in the current one). Using the projection derivatives and the rotation calculated with respect to the previous frame, we understand how a change of one pixel location is translated to the current frame and by doing so we find the relevant scale level in the pyramid.

3. Now we get to the actual search part and we try to find some points of the PVS in the image. The search in done by calling the Tracker::SearchForPoints() method twice – first, as a coarse search [G34] and then as a finer search [G38]. The main difference between the two searches is that in the coarse tracking stage, we search only for corners in the low resolution levels of the pyramid, while in the finer search stage we search for corners from all the pyramid levels. The Tracker::SearchForPoints() tries to find each point by finding a match to the patch around it - PatchFinder::FindPatchCoarse() [G35]:
   It looks at the appropriate level of the target key frame in order to try and find the patch template in it by performing a cross correlation (zero-mean SSD) between a patch around a corner in the new frame and a patch around a corner in the existing key frame in order to find a match between the corners. This is done only for FAST corner points which are within a defined search radius.
   Then we try to get a sub pixel position of the matched corners using the PatchFinder::IterateSubPixToConvergence() method [G36].

   Afterward, if we have found enough matches (with respect to gvnCoarseMin, which is elaborated in the configuration part) - we can perform a Gauss-Newton convergence process [T12] in order to update the pose estimation of the current-frame (Tracker::CalcPoseUpdate()) [G37]. When updating the pose, we also mark the outliers from the convergence process (which we will observe later on in the Mapping section).
   After we did the coarse search stage, we can perform the finer search stage –
   First, we can use the estimated camera model (from the coarse stage) in order to estimate the location of each point from the key frame in the current-frame. Then we repeat the Tracker::SearchForPoints() step and the Gauss-Newton convergence process [G41] in order to update the pose estimation of the current-frame. As we have mentioned before, the finer search in done for corners from all the pyramid levels, but is limited to a maximum amount (MaxPatchesPerFrame, which is elaborated in the configuration part).

4. Finally, we save the results of the current tracking and find the mean scene depth from the tracked features [G42]. The scene depth will be used shortly in order to update the motion model, and is also used for the epipolar search (as we have seen in the map initiation part).

After the Tracker::TrackMap() is done, we update the velocity in the motion model according to the estimated camera model from the tracking stage - Tracker::UpdateMotionModel() method [G43]. Also, we calculate the velocity with respect to the mean scene depth (from the previous stage), and we will use it in order to decide if we should perform a coarse tracking stage in the next frame or not. Note that we can tolerate more translational velocity when far away from scene.

The final step in this tracker branch is to assess the tracking quality – which is done in the method Tracker::AssessTrackingQuality() [G44].
Based on the percentage of points that were matched out of the whole points that we tried to find (in step 3 of the Tracker::TrackMap() method), we decided if the tracking quality is good.
If it's bad, we update the number of lost frames (which is used in order to decide if we are lost, as we have seen above).

In Figure 4 we can observe the map that was created so far (the red dots) and the drone positions along the flight (the green trail). A red cross marks the current pose of the drone with in position coordinated and orientation written in the buttom.

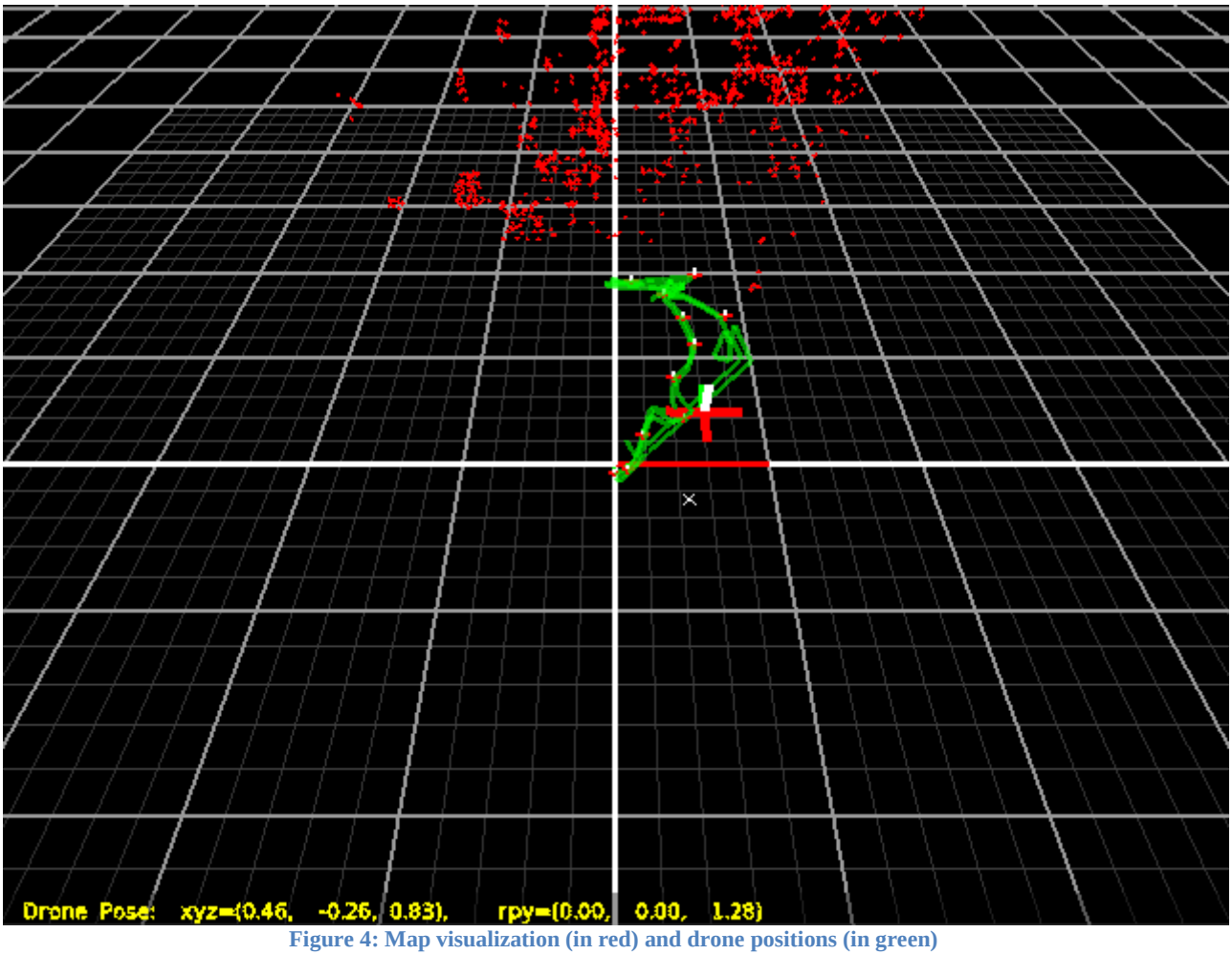


Figure 4: Map visualization (in red) and drone positions (in green)

Now, as you can recall, we have entered the second Tracker branch under the condition that we are not lost [G27]. But if we are lost, meaning the tracking has failed – we attempt to recover by using the method Tracker::tryToRecover() [G28].

In this attempt, if we have a map – we can attempt recovery by going over all the key frames and compare them to the current frame using zero-mean SSD.
The key frame with the highest score is then used when calculating the rotation between it and the current frame (the same way we did it in the beginning of the second tracking branch).
Now we have a new estimation for the current frame model, and we can repeat the Tracker::TrackMap() stage, but this time we force the coarse stage to take place.

2.3 Mapping

This part of the PTAM is performed by MapMaker::run(), the basic method of the thread [G45]. The Mapping stage handles the queue of key frames that the Tracker filled up – Adding them into the map and improving the map while doing so. The Mapping logic handles two cases – an empty queue and waiting key frames in the queue.
Note that most of the method in this part we have already reviewed in the previous sections.

Mapping Logic
First, we'll address the case where the queue of key frames from the Tracker is empty, so the Mapper thread can exploit it to improve the map with the existing key frames –

1. The first step to improve the map is to perform a local BA – by calling the MapMaker::BundleAdjustRecent() method [G48]. Meaning, we find all the map points only from the current frame and its 4 nearest neighbors in the map. And then we perform BA using these points with all the key-frames these points appear in.
   Every BA is done using the basic Mapper method - MapMaker::BundleAdjust().

2. If the local BA reached convergence, we can add more observations for points on existing key frames in the map, by calling MapMaker::ReFindNewlyMade() [G49]. Again, this is similar to Tracker::SearchForPoints().

3. Also if the local BA convergence, we can now perform a global BA - by calling the MapMaker::BundleAdjustAll() method [G50]. Meaning, we perform BA on all the map points with all the existing key-frames using MapMaker::BundleAdjust(). This step requires heavy computation and a long processing time that grows as the map expands.

4. Last, we can try to re-find measurements that were marked as outliers by the tracker (in the pose solving stage) by calling the MapMaker::ReFindFromFailureQueue() method [G51]. Again, this is similar to the previously reviewed method - Tracker::SearchForPoints(). Note that this step happens in a very low probability.

Now, whether the queue is empty or not, we delete the points that where marked as outliers by the tracker (in the pose solving stage) by calling the MapMaker::HandleBadPoints() method [G46]. As we have seen in the 4th step above, we first try to fix these outliers, and if not succeeded we can delete them at this point.

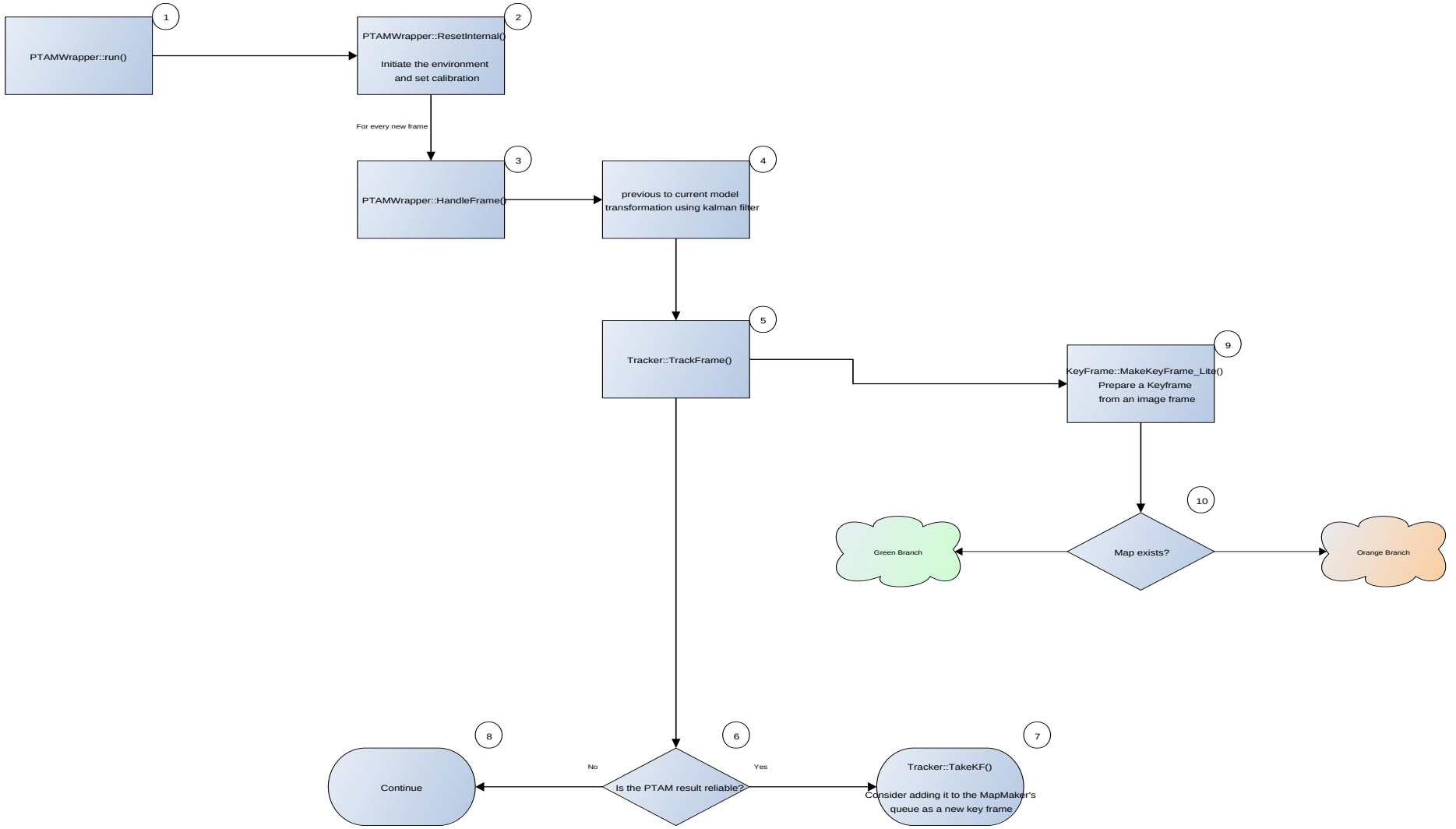
The second case, as we have mentioned, is the case where the Mapper queue isn't empty and we call the method MapMaker::AddKeyFrameFromTopOfQueue() [G52] in order to add the oldest key frame that the Tracker added to the queue –

1. First, the method MapMaker::ReFindInSingleKeyFrame() [G53] is called. Basically, for every point in the map we try to find a match for it in the current frame. This is done much like the tracker did when using Tracker::SearchForPoints().

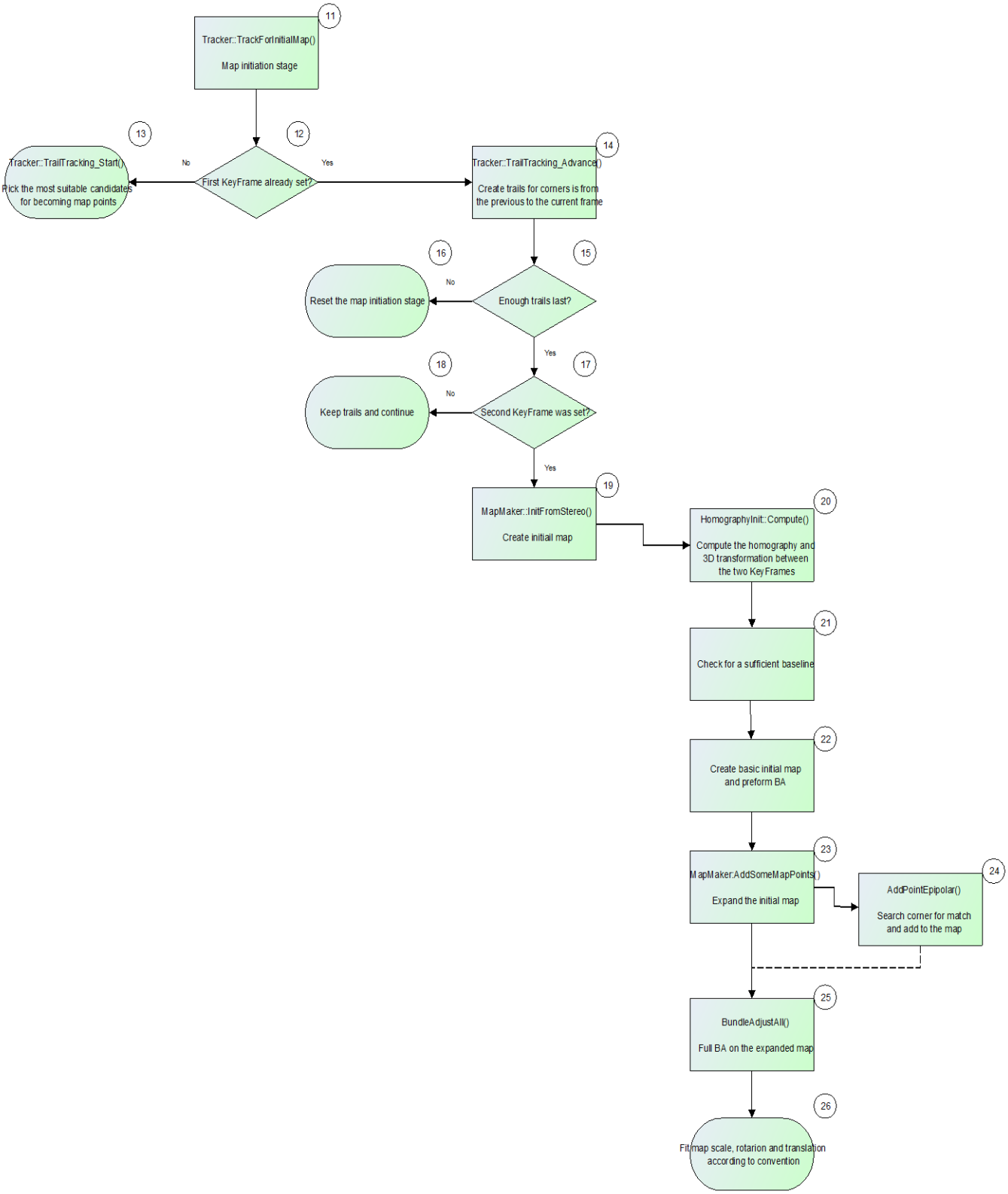
2. Second, we want to expand the map by adding points using an epipolar search on all the pyramid levels of the current key frame. This step is similar to the one we have seen in the map initiation stage, and again we use the MapMaker::AddSomeMapPoints() method [G54].

2.4 PTAM Flow Charts

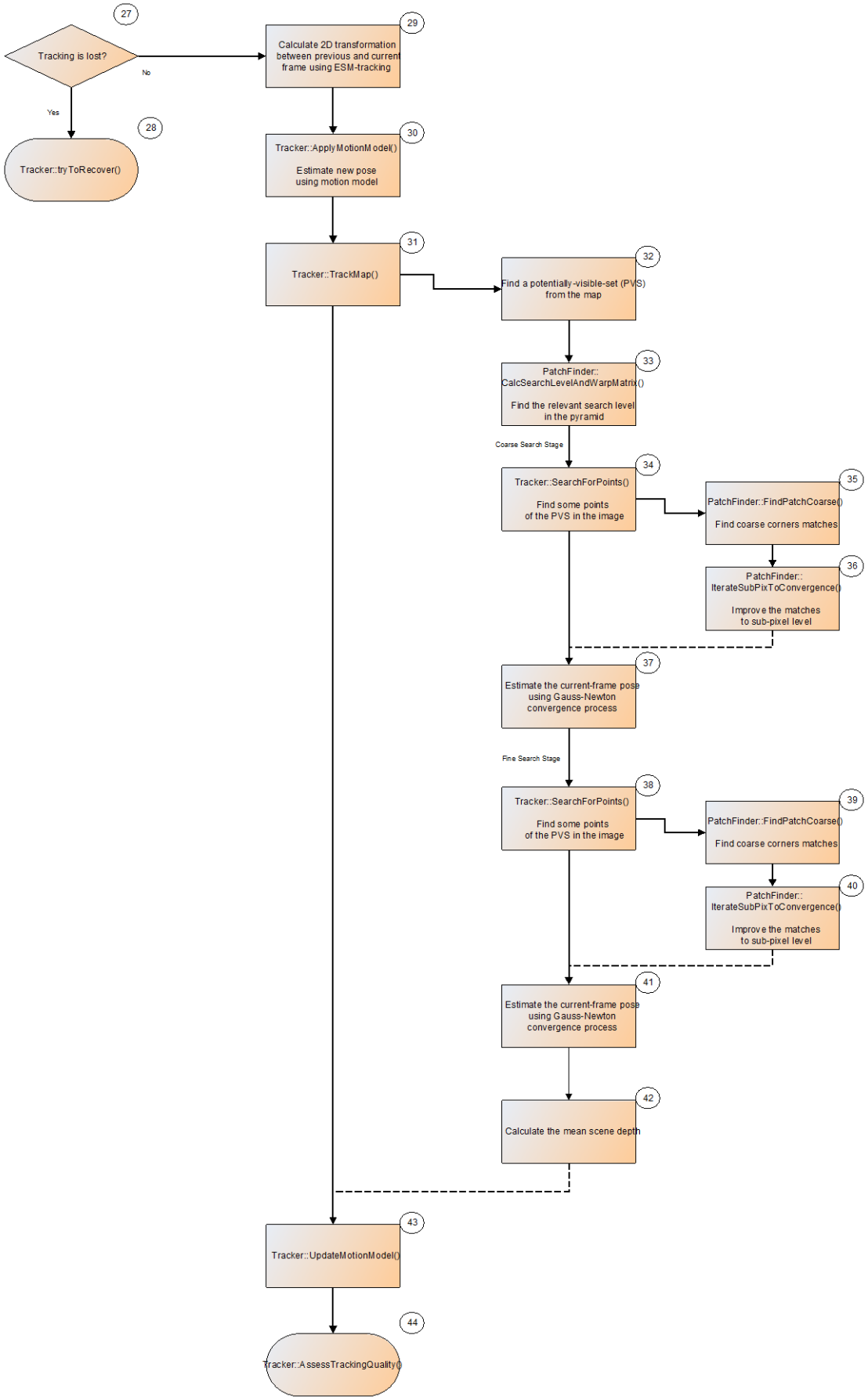
- **Tracking**



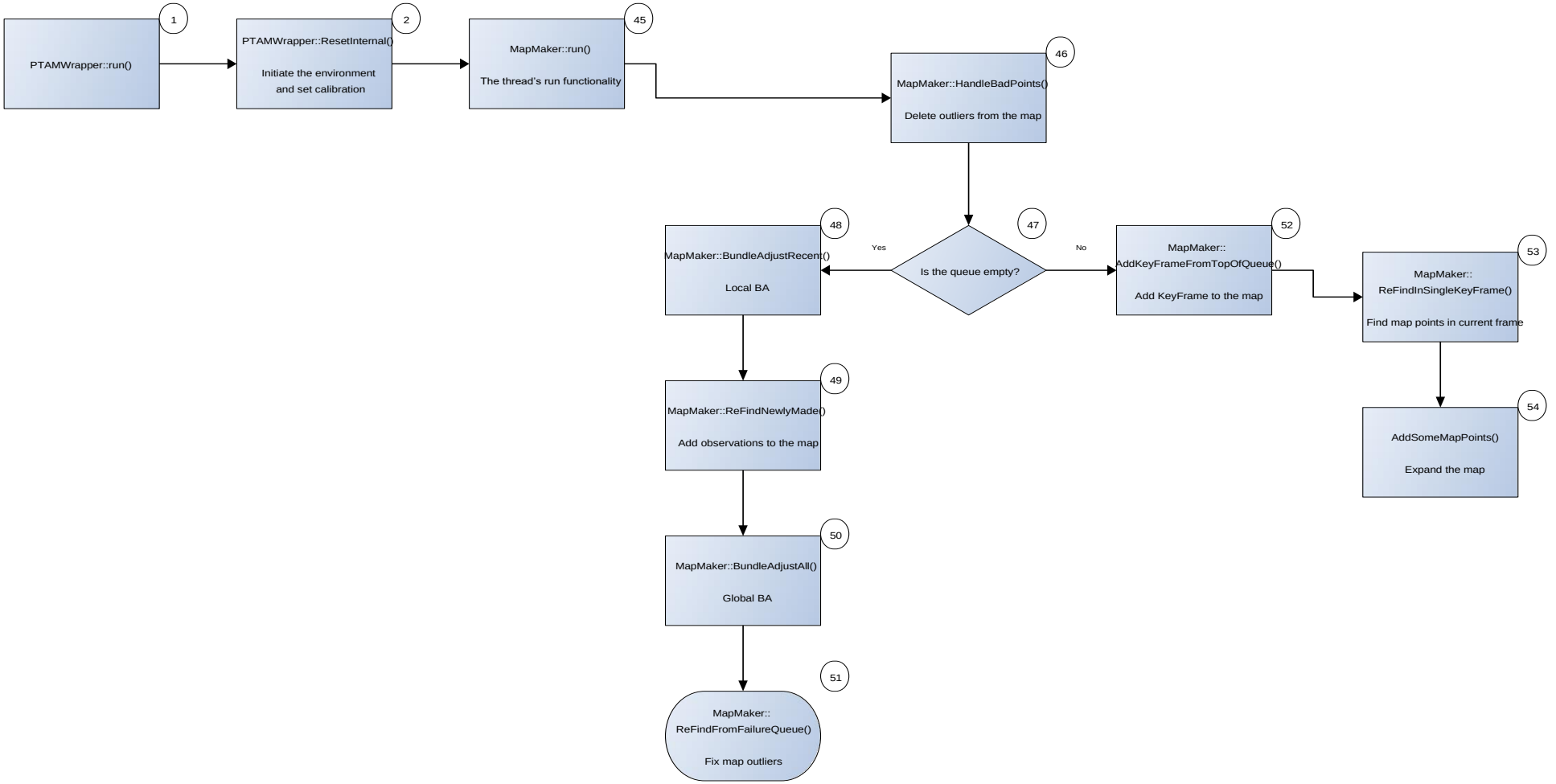
- **Tracking – Green Branch**



- **Tracking – Orange Branch**



- **Mapping**


1
PTAMWrapper::run()
2
PTAMWrapper::ResetInternal()
Initiate the environment and set calibration
45
MapMaker::run()
The thread's run functionality
46
MapMaker::HandleBadPoints()
Delete outliers from the map
47
Is the queue empty?
Yes
No
48
MapMaker::BundleAdjustRecent()
Local BA
52
MapMaker::
AddKeyFrameFromTopOfQueue()
Add KeyFrame to the map
53
MapMaker::
ReFindInSingleKeyFrame()
Find map points in current frame
54
AddSomeMapPoints()
Expand the map
49
MapMaker::ReFindNewlyMade()
Add observations to the map
50
MapMaker::BundleAdjustAll()
Global BA
51
MapMaker::
ReFindFromFailureQueue()
Fix map outliers

3 Background Terms

1. Kalman Filter

The Kalman filter is a well-known method to filter and fuse noisy measurements of a dynamic system to get a good estimate of the current state. It assumes that all observed and latent variables have a (multivariate) Gaussian distribution, the measurements are subject to independent, Gaussian noise and the system is linear. Under these assumptions it can even be shown that the Kalman filter is the optimal method to compute an estimate of the current state as well as the uncertainty (covariance) of this estimate.
In our case, the extended version of the Kalman filter was used. This version drops the assumption of a linear system, making it applicable to a much wider range of real-world problems.
The estimated state in this case includes 10 parameters that express the location (3), the velocity in each axis (3), the orientation angles (3) and the yaw-rotational speed (1).
For further information please refer to sections 4 and 7.4 in the thesis [8].

2. FAST (Features from Accelerated Segment Test) corner detection algorithm [10]

The segment test criterion operates by considering a circle of sixteen pixels around the corner candidate p. The basic logic is to classifies p as a corner if there exists a set of n contiguous pixels in the circle which are all brighter than the intensity of the candidate pixel Ip plus a threshold t, or all darker than Ip – t.
In our case n equals 10, meaning, 10 contiguous pixels in the circle are needed as explained above.
This algorithm was chosen for the PTAM implementation, since it is much faster than other corner detectors (such as SIFT, Harris, etc.) and is capable to process a real time frame rate stream. In addition, this algorithm produces corners of high quality, despite its speed.

3. Non-Maximal Suppression

Non-maximum suppression is method to find "the largest" edge.
After applying gradient calculation, the edge extracted from the gradient value is still quite blurred. Thus, non-maximum suppression can help to suppress all the gradient values (by setting them to 0) except the local maxima, which indicate locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:
Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions.
If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (i.e., the pixel that is pointing in the y-direction, it will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed.
Note that there are many implementations for this concept.

4. Shi-Tomasi Corner Detector [11]

This corner detector is similar to the Harris corner detector, with the main difference being its scoring function.

As in the case of the Harris corner detector, for each corner we want to observe it's A matrix:

$$A = \sum_{u}\sum_{v} w(u,v) \begin{bmatrix} I_x(u.v)^2 & I_x(u.v)I_y(u.v) \\ I_x(u.v)I_y(u.v) & I_y(u.v)^2 \end{bmatrix}$$

Where w(u,v) is the filter window around the pixel, and Ix, Iy are the partial derivatives of the pixels values.

In the case of Shi-Tomasi corner detector, the scoring for each corner is given by observing the minimal eigenvalue of the matrix A. The corners with the highest scores are the most significant ones.

5. Homography

2D homography is also known as a projective transformation.

A 2D homography is an invertible mapping h from P2 to itself such that three points x1,x2,x3 lie on the same line if and only if h(x1),h(x2),h(x3) do.

In other words, it's a line preserving transformation between two images trough a planar surface.

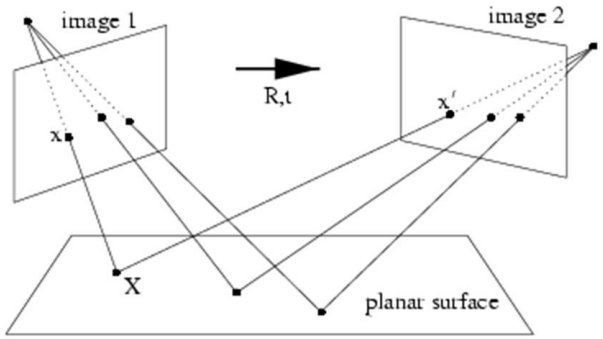


Figure 5: The homography transformation between two images through a planar surface

6. Random sample consensus (RANSAC)

Random sample consensus (RANSAC) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method. It is a non-deterministic algorithm in the sense that it produces a reasonable result only with a certain probability, with this probability increasing as more iterations are allowed.

In our case, we want to estimate a homography from the matching points in two frames. The less outliers we have and the more iterations we allow – the probability for finding the accurate homography increases.

7. Inertial Measurement Unit (IMU)

An inertial measurement unit (IMU) is an electronic device that measures and reports a body's specific force, angular rate, and sometimes the magnetic field surroundings the body, using a combination of accelerometers and gyroscopes, sometimes also magnetometers.

In our case, we use the IMU in order to estimate the translation and orientation change of the drone.

8. Triangulation

Triangulation is the principle used in photogrammetry to produce 3-dimensional point measurements. By mathematically intersecting converging lines in space, the precise location of the point can be determined. These lines are the lines of sight from the pixel in the sensor plane to the point in the world.

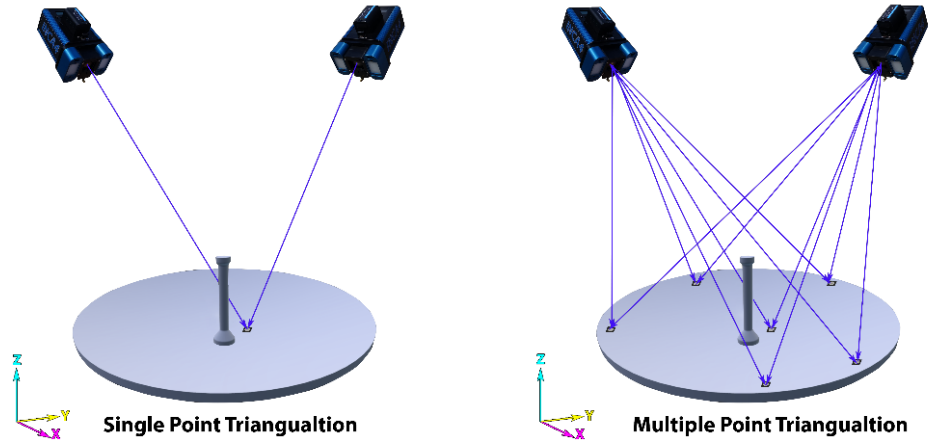


Figure 6: On the left - a single point triangulation. On the right - a muliple point triangulation

9. Bundle Adjustment

Given a set of images depicting a number of 3D points from different viewpoints, bundle adjustment can be defined as the problem of simultaneously refining the 3D coordinates describing the scene geometry, the parameters of the relative motion, and the optical characteristics of the camera(s) employed to acquire the images, according to an optimality criterion involving the corresponding image projections of all points, for instance minimizing the reprojection error between the image locations of observed and predicted image points.
In other words, when we have a set of images taken from different locations with different orientation, we can estimate their pose by calculating the location of each feature point from at least two images, and improve the pose estimation according to the reprojection error.

10. Epipolar Search

An epipolar search is a search along the epipolar line.
Given two images taken from different locations, the point on the image that marks the direction to the other image is called an epipole (marked by e in the figure below).
The epipolar line is the line in the image (the red line in the figure below) connecting the epipole to the pixel where the feature is seen (marked by XR/XL in the figure below).
If a feature is seen in one image, we can know the line of sight it lies on, but we don't know the depth of its location. Here, an epipolar search becomes useful, since this line of sight is translated to a line in the second image, meaning the wanted feature must be found along it.

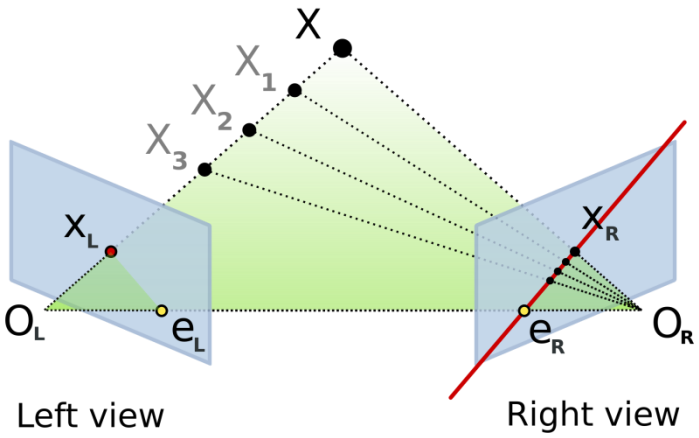


Figure 7: An epipolar geometry – the epipoles are denoted by e, and the epipolar line is marked in red

11. ESM-tracking a la Benhimane & Malis [12]

This tracking method contains two steps – finding a homography and translating it into a new pose.
When finding the homography, we suppose that the object we aim to track is planar. Since the object is supposed to be planar, there is a homography that transforms each pixel of the reference pattern into its corresponding pixel in the current image.
The second step is a visual servoing method that does not need any measure of the structure of the observed target. In this method, an isomorphism is defined between the camera pose and the visual information extracted from the reference image and the current image only. Given this isomorphism, a stable control law is used, which also relies on visual information only.
According to this method, the homography is translated into the camera translation velocity and the camera rotation velocity, which in their turn translated into the relative translation and rotation from the reference image.

12. Gauss–Newton algorithm

The Gauss–Newton algorithm is used to solve non-linear least squares problems in an iterative manner. It is a modification of Newton's method for finding a minimum of a function. Unlike Newton's method, the Gauss–Newton algorithm can only be used to minimize a sum of squared function values, but it has the advantage that second derivatives, which can be challenging to compute, are not required.
In our case, we want to minimize the reprojection error while the Jacubian matrix contains the partial derivatives of the XYZ location to the IJ pixel coordinates.

4 PTAM Configuration

Here we will review a list of parameters that affect the PTAM algorithm and can be controlled easily.
Note that when referring to the configuration files, we refer to these files:

1. tum_ardrone → cfg → StateestimationParams.cfg
2. tum_ardrone → src → stateestimation → PTAM → settingsCustom.h

- **minKFTimeDist**

The minimum required time (in seconds) from the last key-frame that was added to the map.
Is used in the Tracker when considering an insertion of a new key-frame.
Originally set to 0.5 in the configuration file. We should consider increasing it if the scene changes very slowly or if we expect to build a large map that will require a lot of key-frames.

- **minKFDist**

The minimum required distance (in meters) from the closest key-frame in the map.
Is used in the MapMaker when deciding whether to add a new key-frame or not.
Originally set to 0.4 in the configuration file. We should consider increasing it if the scene is relatively far and we expect that only after a significant movement of the drone we will notice a change in the scene.

- **minKFWiggleDist**

The minimum required ratio of the distance from the closest key-frame in the map and to the mean scene depth.
Is used in the MapMaker when deciding whether to add a new key-frame or not.
Originally set to 0.075 in the configuration file. This parameter is similar to minKFDist, but is more robust since we don't need any prior knowledge of the scene dimensions.

Figure 8 demonstrates the effect of the minKFWiggleDist parameter. In this experiment we set its value to 7.5, significantly higher than its default value.
In the figure, we can see 3 frames taken from the video stream according to their appearance in the video (from top to bottom). The first frame shows that the Map covers this area well, since keyFrames that cover it were taken. The second and third frames show that the area on the right side is not covered in the Map, and due to the modification we performed in minKFWiggleDist, the PTAM algorithm cannot add a new keyFrame to cover this area (since the existing KeyFrames are located to close to it, with respect to minKFWiggleDist).

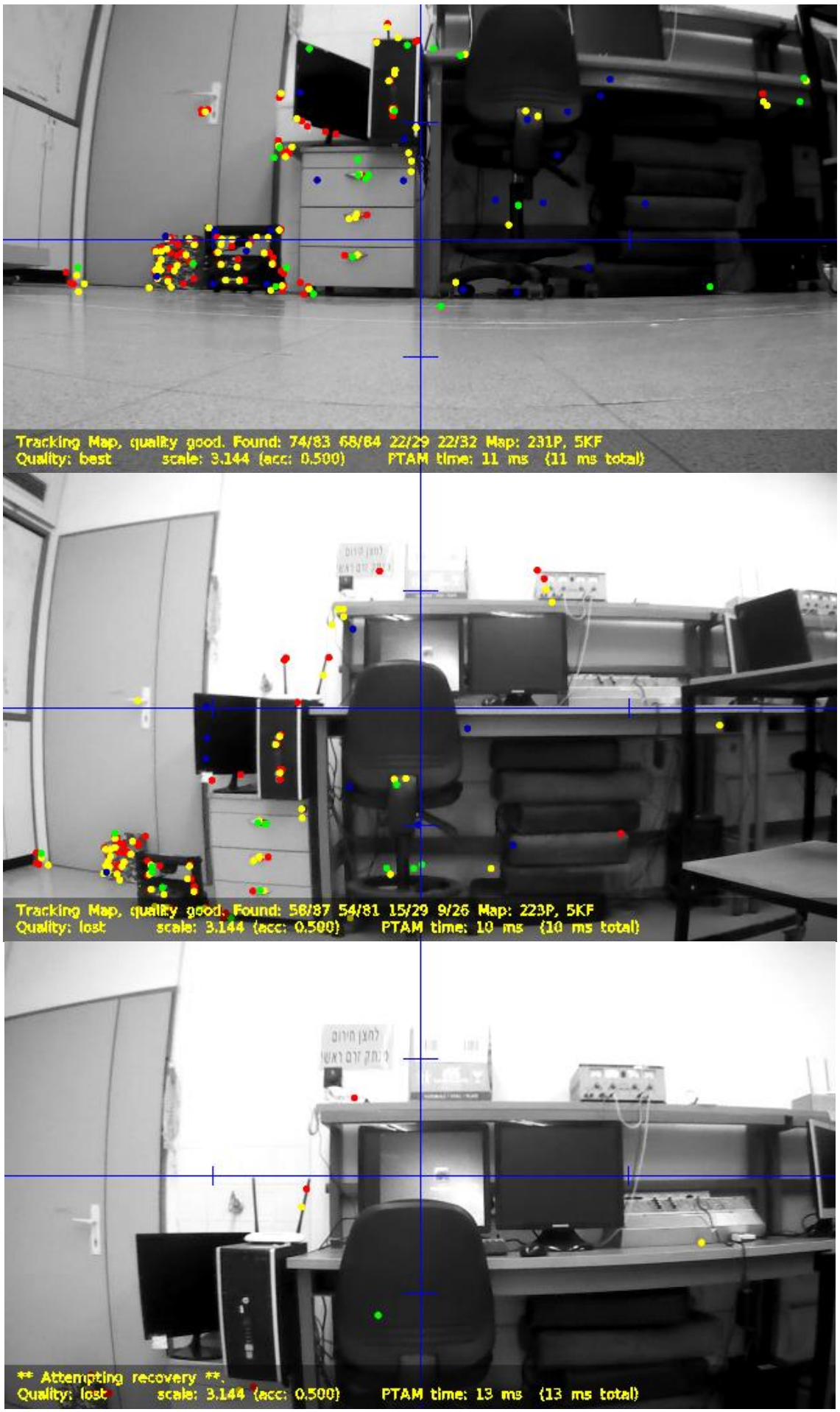


Figure 8: PTAM result visualization on 3 frames of the video stream with Minimum KeyFrames Scale set to 7.5

- **maxKF**

The maximum amount of key-frames allowed in the map.
Originally set to 200 in the configuration file. We should consider increasing it if we expect a large map or a scene that changes significantly over time. We should consider decreasing it if we want to improve the BA performance. Note that changing this parameter is risky and it's better to control the insertion new key-frames over limiting the total amount.

Figure 9 demonstrates the effect of the maxKF parameter. In this experiment we set its value to 2 frames, significantly lower than its default value.
As we can see in low bar of the captured frame, the Map contains only 179 points and 2 KeyFrames (as expected). For comparison, we can observe Figure 2 that show the scene with this parameter set to its original value. This modification derives a poor coverage of the scene, which is visible also in the map view.

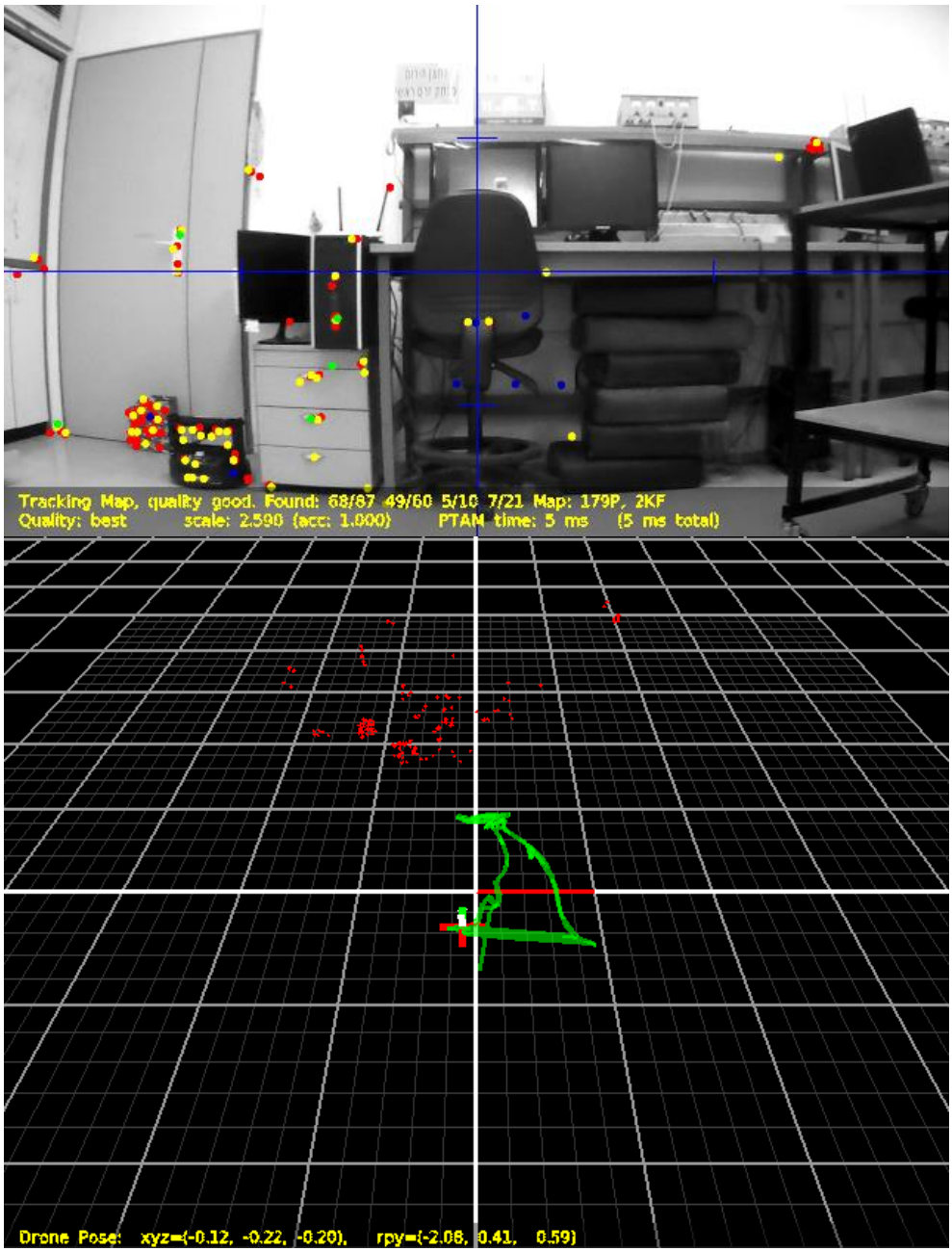


Figure 9: PTAM result visualization on the video stream (up) and PTAM map visualization (down) with Maximun KeyFrames set to 2 frames

- **FRAMES_BETWEEN_KEYFRAMES**

The minimum required frames to pass before inserting a new key frame to the map.
Originally set to 10 frames. This parameter allows us to control the insertion rate of the key frames and the density of the map. When working on a low FPS stream it is necessary to reduce it.

- **LEVELS**

The number of levels in the Gaussian pyramid that is created for every frame.
This parameter is defined in the KeyFrame header file and set to 4. we should consider increasing it in case of high image dimensions, but this may also be followed by not using all the pyramid levels. We should consider decreasing it in case of low image dimensions or when trying to decrease the time spent on handling each frame (on the possible expense of damaging the performance).

- **TRACKER_COARSE_MIN_DEFAULT**

The minimum number of large-scale features that is required for optimizing the pose in the coarse stage.
Originally set to 20 features. Increasing it means that the update after the coarse stage will be more reliable, but may be available for fewer frames.

- **TRACKER_COARSE_MAX_DEFAULT**

The maximum number of multi scale features in the coarse stage.
Originally set to 60 features. Increasing it means that the update after the coarse stage will be more reliable, and also more accurate (since we will use more small scale features), but will take more processing time. Decreasing it applies the opposite meaning.

- **TRACKER_COARSE_RANGE_DEFAULT**

The pixel search radius for find a patch match for the coarse features.
Originally set to 30 pixels. Deceasing it will speed up the process, but may lead to less matches. Increasing it applies the opposite meaning.

Figure 10 demonstrates the effect of the TRACKER_COARSE_RANGE_DEFAULT parameter. In this experiment we set its value to 2 pixels, significantly lower than its default value.
As we can see, our Map contains 4 KeyFrames, but only 125 points, since the PTAM couldn't find a match for the features and add it to the map later on. Lowering this parameter caused the algorithm to find less matches, since its search area was too limited.

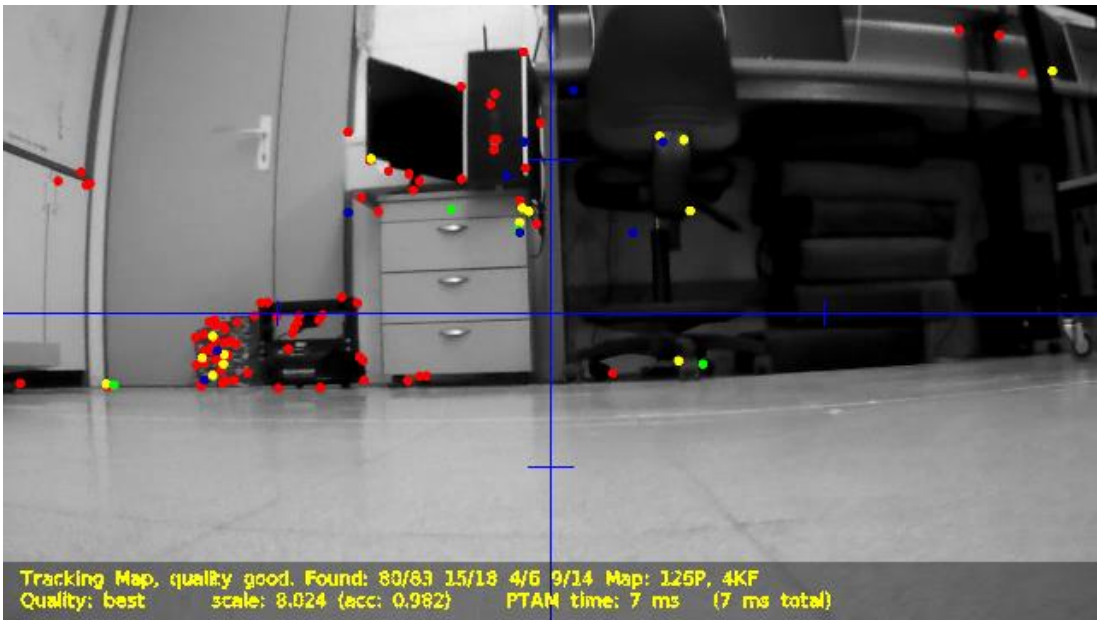


Figure 10: PTAM result visualization on the video stream with Coarse Range set to 2 pixels

- **TRACKER_MAX_PATCHES_PER_FRAME_DEFAULT**

The maximum number of patches created from each frame in the points matching step. Originally set to 1000. Increasing it means more attempts to match the points will happen and the probability to find a match will increase (on the expense of processing time).

Figure 11 demonstrates the effect of the TRACKER_MAX_PATCHES_PER_FRAME_DEFAULT parameter. In this experiment we set its value to 50 patches, significantly lower than its default value.
In the figure, we can see 2 frames taken from the video stream according to their appearance in the video (from top to bottom). The first frame shows that we were able to locate only few points of the Map, since only 50 patches were allowed to be searched. The second frame shows again that we were able to locate only few points of the Map, but we can also notice that points that appear in the first frame were not found in the second one (again due to the modification we have made).

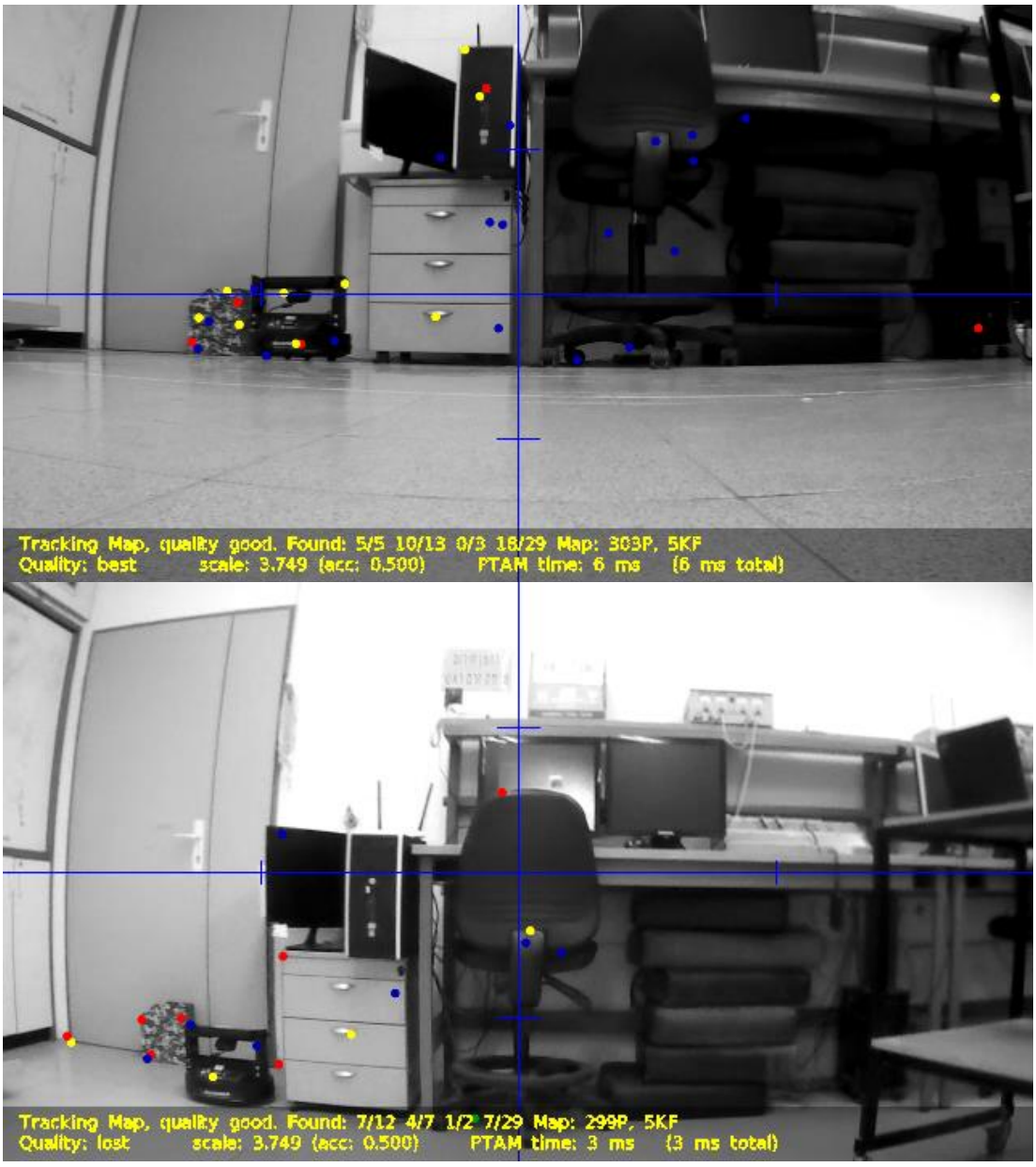


Figure 11: PTAM result visualization on 2 frames of the video stream with Maximun Patches Per Frame set to 50 patches

- **TRACKER_COARSE_SUBPIXELITS_DEFAULT**

The maximum sub-pixel iterations for the convergence process in the coarse features stage.
Originally set to 4 iterations. Increasing it will lead to more accurate results for the coarse stage, over the expense of processing time. Decreasing it is not recommended.

- **TRACKER_COARSE_DISABLE_DEFAULT**

A control flag that disables coarse stage logic when set to 1 (except after recovery).
Originally set to 0. Disabling this stage will save processing time, but will cause to more tracking failures.

- **TRACKER_COARSE_MIN_VELOCITY_DEFAULT**

The minimum velocity (in the map world with respect to the scene depth) above which the coarse stage is used.
Originally set to 0.006. Increasing it will reduce the processing time, but is very delicate and not recommended.

- **BUNDLE_MAX_ITERATIONS**

The number of allowed iterations in the Bundle Adjustment process. Note that usually the BA won't require the maximum iterations in order to converge.
Originally set to 20 iterations. This step is very important while building the map, and is directly correlated to the quality of the map and the entire process. Nevertheless, increasing it significantly in order to improve the mapping is not recommended, since it is a major time consumer.

5 Installation

Hereby listed are the installation instructions for the PTAM algorithm package and the code environment needed for debugging and changing the PTAM itself.

1. Install the tum_ardrone package:

    cd catkin_ws/src

    git clone https://github.com/rdelgadov/ -b kinetic-devel

    cd ..

    rosdep install tum_ardrone

    catkin_make

2. Install Eclipse (photon version)

3. Build the project by running in ~/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

4. Build the debug mode for the project by running in ~/catkin_ws/build$:

    cmake ../src -DCMAKE_BUILD_TYPE=Debug

    source ~/catkin_ws/devel/setup.bash

5. Start the Eclipse

6. In the Eclipse:

    File --> Import --> General --> Existing Projects into Workspace

    Browse for your package's directory (select the build directory). Do NOT select Copy projects into workspace.

6 Debug

Hereby listed are the debug instructions for the PTAM algorithm package using the Robot Operating System (ROS):

1. On terminal 1: roslaunch tum_ardrone tum_ardrone.launch
2. When running on a real time drone stream –
   On terminal 2: roslaunch tum_ardrone ardrone_driver.launch
3. When running on a pre recorded drone stream (for instance: "calibFlight.bag") –
   On terminal 2: rosbag play -l calibFlight.bag –pause
4. Open main_stateestimation.cpp in the eclipse
5. Place the break points in the eclipse environment
6. Run the eclipse in debug mode
7. Continue the run (so the eclipse is waiting for the video stream)
8. Press the space bar in terminal 2 in order to un-pause the rosbag
9. In the drone video gui initiate the process by pressing the space bar once
10. In the drone video gui at this point a trail for each feature point will appear
11. In the drone video gui continue the initiation by pressing the space bar again after a noticeable movement in the video
12. When you reach a break point – you can pause the rosbag by pressing the space bar in terminal 2 (and then returning to the eclipse for further debug)

Additional notes:

1. You can slow the rosbag publish rate by adding “-r” flag to the play command followed by the new rate factor, for example:
   rosbag play -l calibFlight.bag -pause -r 0.1

2. When debugging the code without meanwhile pausing the rosbag, an inconsistency in the telemetry may occur and cause an unexpected behavior.

3. When modifying the code or the configurations, it is recommended to terminate all sessions and restart the debug steps from the top.

7 Discussion

The PTAM algorithm was one of the first SLAM algorithms that allowed real time performance with impressive accuracy. As such, it is considered a well founded base for other SLAM algorithms. We believe that for mastering any latest state of the art SLAM algorithms, one must first understand the many foundations that the PTAM implements and focuses on. For that end, choosing the PTAM as an analysis target was justified.

One of the leading SLAM algorithms is the ORB-SLAM, which is quite similar to PTAM, yet attains more impressive performance in practice. It has 4 main improvement compared to PTAM. First, in addition to the PTAM threads (tracking and mapping), it implements a third thread for loop closing detection, in order to achieve consistent localization and mapping. Second, it has an automatic map initialization with a model selection on two paralleling thread calculating Homography and Fundamental ego-motion with RANSAC, while PTAM requires manual operation to finish initialization. Third, it uses ORB feature detector and descriptor instead of image patches used in PTAM, improving robustness of image tracking and feature matching under scale and orientation change. Fourth, is uses multi-scale mapping, including local graph for pose bundle adjustment, co-visibility graph for local bundle adjustment, and essential graph for global bundle adjustment after loop closure detection.
ORB-SLAM2 [13] implements the same logic as the ORB-SLAM, with several modifications in order to handle stereo and RGB-D cameras as well.

Another impressive SLAM algorithm is the LSD-SLAM (Large-Scale Direct Monocular SLAM), which is a bit different from PTAM or ORB-SLAM, yet its main steps remain similar. It should be classified into so-called direct methods, that is, doing state estimation based on image pixels directly, rather than relying on image features. The tracking is performed by image alignment using a coarse-to-fine. Depth estimation is just like many other SLAM systems, using an inverse depth parameterization with a bundle of relatively small baseline image pairs. Map optimization is also executed in commonly used graph optimization, with existing keyframe poses expressed just like ORB-SLAM does. LSD-SLAM actually recovers a ‘semi-dense’ map, since it only estimates depth at pixels solely near image boundaries, which may be the main reason for it to be the first direct visual SLAM system that can run in real-time on a CPU.

Future work that could continue our work with respect to drone applications, should focus on the ORB-SLAM algorithm and its differences from the PTAM algorithm. While the ORB-SLAM improves the PTAM logic, it could come on the expense of performance in some cases. For that end, it is important to understand the strengths and weaknesses of each of them and to create an application based logic that optimizes them both.

8 Conclusion

In this work we have focused on analyzing the parallel tracking the mapping (PTAM) algorithm implementation as an example of a simultaneous localization and mapping (SLAM) algorithm that is commonly used in drones' applications.

Our main goal was to provide a theoretical and practical overview of the algorithm in order for the reader to understand the PTAM logic, its limitations and the ways to modify it for future use.

First, in order to achieve that, we have reviewed the literature regarding the PTAM algorithm and it's like.
Second, for effectively handling the PTAM specific implementation in the thesis, we have focused on learning the relevant code environment and services that were used - Robot Operating System (ROS).
Third, we examined several debug methods, for choosing one that will be the most suitable for debugging and learning the PTAM implementation for a drone data stream.
Fourth, we mapped and analyzed the PTAM implementation and created an informative and practical documentation for this end.

By adding a flow chart that details the implementation logic in a graphic manner, we hope to ease on the reader in his learning process of the algorithm.
Furthermore, for allowing readers who are new to the field to better understand our work, we have gathered relevant terms that are essential for this purpose.

In summary, after reviewing this work, the reader is able to follow the PTAM logic, to install and debug its code and to understand how to utilize and modify it for his use.

9 Bibliography

[1] G. Klein and D. Murray. Parallel tracking and mapping for small AR workspaces. In Proc. of the International Symposium on Mixed and Augmented Reality (ISMAR), 2007.

[2] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: An efficient alter- native to sift or surf. In 2011 International Conference on Computer Vision, pages 2564–2571, Nov 2011.

[3] Lowe, David G. (2004). Distinctive image features from scale-invariant key points. International Journal of Computer Vision 60(2): 91-110.

[4] H. Bay, T. Tuytelaars, and L. Van Gool. SURF: Speeded up robust features. In ECCV, 2006.

[5] G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. g2o: A general frame- work for graph optimization. In IEEE International Conference on Robotics and Automation, 2011.

[6] R. Mur-Artal, J. M. M. Montiel, and J. D. Tards. Orb-slam: A versa- tile and accurate monocular slam system. IEEE Transactions on Robotics, 31(5):1147–1163, Oct 2015.

[7] J. Engel, T. Sch¨ops, and D. Cremers. Lsd-slam: Large-scale direct monocular slam. In Computer Vision–ECCV 2014, pages 834–849. Springer, 2014.

[8] J. Engel, “Autonomous Camera-Based Navigation of a Quadrocopter,” Master’s thesis, Technical University Munich, 2011.

[9] Ezio Malis, Manuel Vargas. Deeper understanding of the homography decomposition for vision-based control. [Research Report] RR-6303, INRIA. 2007, pp.90.

[10] Rosten E., Drummond T. (2006) Machine Learning for High-Speed Corner Detection. In: Leonardis A., Bischof H., Pinz A. (eds) Computer Vision – ECCV 2006. ECCV 2006. Lecture Notes in Computer Science, vol 3951. Springer, Berlin, Heidelberg.

[11] Shi, J., Tomasi, C.: Good features to track. In: 9th IEEE Conference on Computer Vision and Pattern Recognition, Springer, Heidelberg (1994).

[12] Benhimane, Selim and Ezio Malis. “Homography-based 2D visual servoing.” Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.

[13] Raúl Mur-Artal and Juan D. Tardós. ORB-SLAM2: an Open-Source SLAM System for Monocular, Stereo and RGB-D Cameras. IEEE Transactions on Robotics, vol. 33, no. 5, pp. 1255-1262, 2017.