ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Autonomous Systems Lab
Prof. Roland Siegwart

Semester Thesis

Visual-Inertial Motion Estimation on Computationally Constrained Platforms

Autumn Term 2014

Supervised by:
Pascal Gohl
Janosch Nikolic
Jörn Rehder

Author:
Marcin Tomasz Dymczyk

Declaration of Originality

I hereby declare that the written work I have submitted entitled

Visual-Inertial Motion Estimation on Computationally Constrained Platforms

is original work which I alone have authored and which is written in my own words.[1]

Author(s)

Marcin	Dymczyk

Student supervisor(s)

Pascal	Gohl
Janosch	Nikolic
Jörn	Rehder

Supervising lecturer

Roland	Siegwart

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (`https://www.ethz.ch/content/dam/ethz/associates/students/studium/exams/files-en/plagiarism-citationetiquette.pdf`). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date

Signature

[1] Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Contents

Abstract

The project presents a concept and implementation of egomotion estimation algorithm that can run online on the computationally constrained platform equipped with Xilinx Zynq-7020 CPU. A minimalistic yet reliable visual odometry algorithm was derived, using BRISK descriptors and Perspective n-Point algorithm paired with RANSAC. The visual-inertial data fusion is realised using Stochastic Cloning Extended Kalman Filter. The algorithm runs on ASLAM Sensor device with framerates reaching 12 Hz and provides satisfactory trajectory estimates.

Symbols

Symbols

$\boldsymbol{\omega}_m$	3-axis gyroscope measurement
$\mathbf{b}_\omega$	gyroscope bias
n_ω	gyroscope noise
$n_{b,\omega}$	gyroscope bias process noise
$\mathbf{a}_m$	3-axis accelerometer measurement
$\mathbf{b}_a$	accelerometer bias
n_a	accelerometer noise
n_a	accelerometer noise
σ_{VO}	visual odometry
$\mathbf{P}$	covariance matrix
$\mathbf{\Phi}$	state transition matrix
$\mathbf{H}$	sensitivity matrix
$\mathbf{R}$	measurement noise covariance matrix
$\mathbf{K}$	Kalman gain matrix
$\mathbf{Q_{cont}}, \mathbf{Q_d}$	noise covariance matrices
$\mathbf{K}$	Kalman gain matrix
$\otimes$	composition of quaternions
$\lfloor \cdot \times \rfloor$	skew-symmetric matrix

Indices

res	residual term
$corr$	correction term
VO	visual odometry
$gravity$	gravity reference, based on accelerometer
c	cloned state/covariance
$cont$	continuous-time
d	discrete-time

Acronyms and Abbreviations

ETH	Eidgenössische Technische Hochschule
ASL	Autonomous Systems Lab
EKF	Extended Kalman Filter
SC-EKF	Stochastic Cloning Extended Kalman Filter
IMU	Inertial Measurement Unit
VO	Visual Odometry
SLAM	Simultaneous localization and mapping
PID	proportional-integral-derivative
UAV	unmanned aerial vehicle
MAV	micro aerial vehicle
FPS	frames per second

Chapter 1

Introduction

1.1 Motivation

The egomotion estimation algorithms are getting more and more attention in the robotics community. Full onboard pose estimation permits different kinds of autonomous systems, such as UAVs, to perform high-level tasks such as exploration, navigation in GPS-denied environments or trajectory following.

So far, most egomotion estimation algorithms, both at ASL and generally, used a high-performance computing platform, often not much weaker than our daily PCs [1][2][3]. Such approach is not efficient taking into consideration weight, size and power requirements of the setup. Especially, it seems not suited at all for MAVs. The idea of the project is to overcome these issues using an already existing device, an ASLAM Sensor.

The ASLAM Sensor is a device equipped with Xilinx Zynq ARM processor, stereo cameras and IMU, weighting only 130 g. So far, it was used at ASL only to stream data to a more powerful platform, for example Intel Atom board [4]. The goal of the project is to use this device to perform full egomotion estimation onboard. Obviously, the lightweight, minimalistic approach introduces some deficiencies. The limited computational power puts some restrictions on algorithm choice, e.g. performing full SLAM seems infeasible and it is necessary to stick to only limited world map or frame-to-frame correspondences. Unfortunately, with such approaches, a temporal drift will not be avoided. On the other hand, successfully running egomotion estimation algorithm on ASLAM Sensor will make it a completely independent solution, ready to be interfaced with any system of choice.

1.2 Related work

The importance of the real-time onboard visual-inertial odometry estimation based on image features was already noticed by Hirschmuller *et al.* [5] and Nister *et al.* [6], who in fact came up with the term *visual odometry*. At the time, there were however no embedded systems that could handle an egomotion estimation task in real-time, and even full-scale PCs struggled to achieve satisfying results. The only possibility to get high performance was to use frame-to-frame matches [7] together with some heuristics and reject more complicated approaches.

Since that time, the quality of estimation has improved significantly, as the newest

real-time solutions are no more limited to simple frame-to-frame correspondences, but can use full potential of SLAM-like approaches and run with satisfactory framerates on a standard PC [2]. The feature matching made a significant step forward and many algorithms can use modern types of descriptors as well as effective prediction of future feature locations [4].

The topic of egomotion estimation is getting more and more attention (e.g. a complete visual odometry tutorial by D. Scaramuzza *et al.* [8] at ETH), but also more work is put into getting portable visual odometry sensors with reasonable estimation quality [3].

Up to 2011, not much work has been done on using visual-inertial odometry on embedded devices – state-of-the-art solutions were both too slow and very expensive. Recently, new technologies were introduced to the market (e.g. common use of FPGAs, ARM NEON engine, multicore ARM CPUs) and the egomotion estimation became feasible. In [9], Goldberg *et al.* present a real-time visual odometry framework running on OMAP3530 chip with high framerates.

This project will deal with the problem in a bit different way, as instead of OMAP processor with DSP core, it will use a Xilinx Zynq processor integrated with FPGA. Visual odometry data will be then fused with IMU measurements.

Chapter 2

Algorithm outline

2.1 Algorithm design decisions

One of the most challenging parts of the project was to decide on approaches, algorithms and general structure of the pipeline. On one hand, it was very important to select methods that can be ran on an embedded system and fit within the desired iteration time (10 Hz rate was considered as acceptable). On the other hand, using too simplistic approaches could most probably negatively affect the overall pose estimation quality.

The visual odometry block was considered to be the most resource-demanding part of the system (and it is indeed true). The search for lightweight, but still reliable visual odometry framework ended up with an algorithm using frame-to-frame matches and estimating the pose using Perspective n-Point method. To make the keypoints matching reliable, binary BRISK descriptors were used [10].

After getting the output of visual odometry block, it would be reasonable to use ASLAM sensor's onboard IMU to additionally improve the pose estimate quality. The filtering technique needs to deal with non-linear systems, but still keep the computational cost minimal. The Extended Kalman Filter seems to be a perfect fit for this task, apart from one important fact. If we consider the visual odometry measurements, they are **not** independent, which violates one of the most important assumptions of EKF. The dependence can be noticed in two ways:

- Visual odometry returns frame-to-frame relative measurements and the final absolute pose estimate will be a concatenation of such transformations – the subsequent absolute poses will become dependent
- Two subsequent visual odometry estimates are dependent, as each one is based on current and previous frame – if we assume that the first estimate uses $\{x_{k-1}, x_k\}$, then the second will use $\{x_k, x_{k+1}\}$ and both will depend on x_k

An answer to this problem is provided by Stochastic Cloning EKF [11, 12], which is specially designed to properly fuse relative measurements.

Stochastic Cloning EKF was previously used for egomotion estimation at ASL and some runtime information data was also available [4]. The performance of full pose filtering using x86 Intel Atom 1.6GHz suggested that the task will not be feasible in prescribed time on ASLAM Sensor. That is the reason why finally attitude-only SC-EKF was implemented, leaving translation unfiltered.

The final high-level pipeline of the algorithm is depcited in figure 2.1.

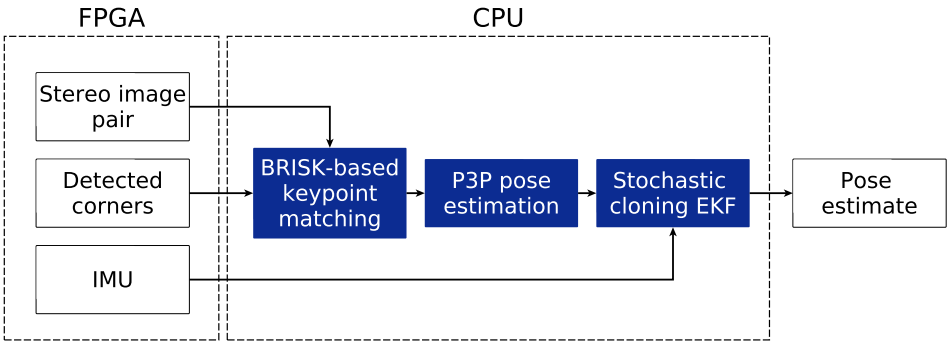


Figure 2.1: High-level view at algorithm pipeline.

2.2 Visual odometry algorithm

2.2.1 Visual odometry pipeline

The visual odometry pipeline is based upon frame-to-frame matching and Perspective n-Point algorithm. The algorithm is kept minimalistic, taking into account main goal of the project, that is running the pose estimation on an embedded device with limited power.

The visual odometry procedure starts with obtaining a stereo image pair and detecting the keypoints. To make matching process easier, we can compute a descriptor for each of the keypoints and then use some dedicated distance function to compare the descriptors (see 2.2.2). The left-right stereo correspondences can be then triangulated (see 2.2.3), so that we obtain a 3D world representation of keypoints. Finally, using previously triangulated keypoints that can be matched to most-recent keypoints, we can compute a relative pose update (see 2.2.4). In the subsequent sections, the theory behind each part of the pipeline as well as detailed design decisions are presented.

The detailed diagram of the visual odometry procedure is presented in the figure 2.2. A simplistic yet intuitive depiction of each step is presented in the figure 2.3.

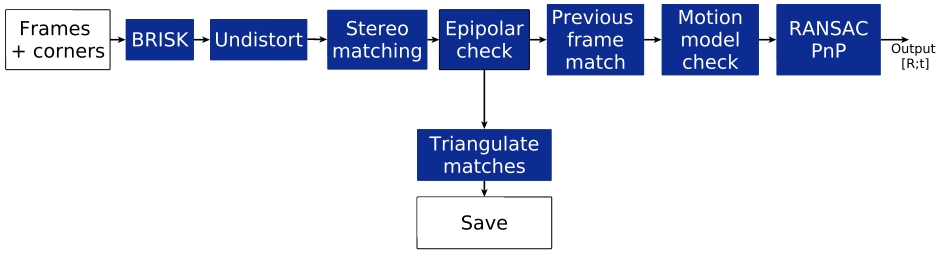
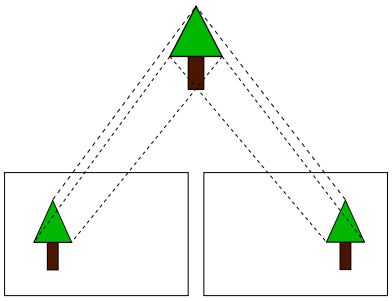
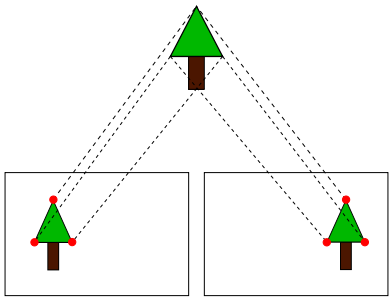


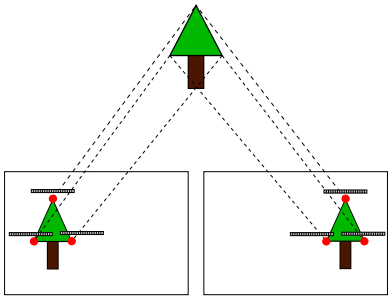
Figure 2.2: Detailed view at visual-odometry pipeline.



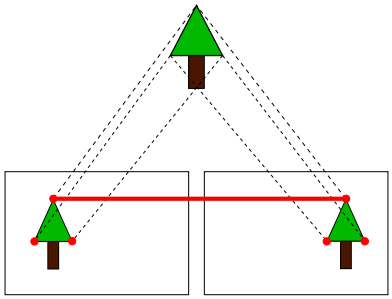
(a) Frames received from FPGA



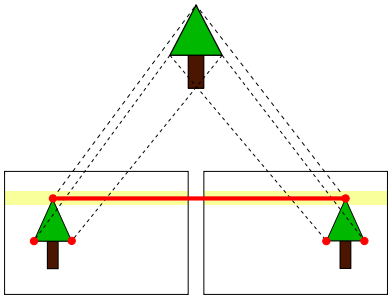
(b) Corners received from FPGA



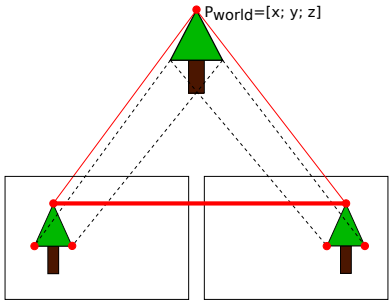
(c) BRISK descriptor extraction



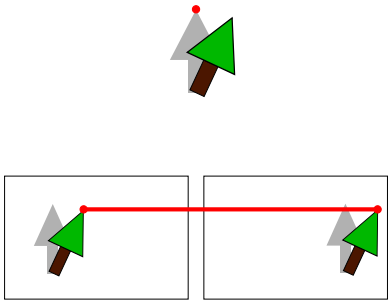
(d) Stereo matching



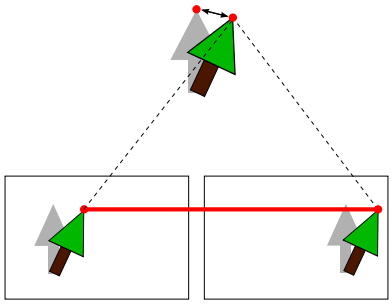
(e) Epipolar constraint check



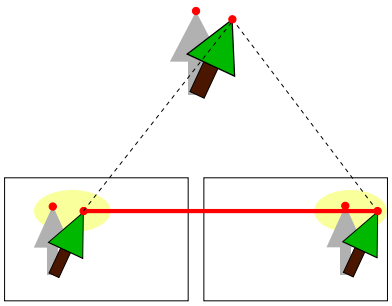
(f) Triangulation



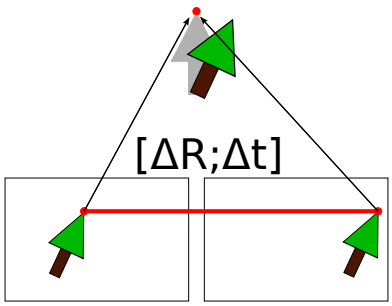
(g) New frames received



(h) Matching to triangulated points



(i) Motion model check



(j) Output pose update by PnP

Figure 2.3: Step-by-step representation of visual odometry pipeline

2.2.2 Keypoints and matching

Keypoint detection

During inital phases of the algorithm development, the keypoints were detected in software using FAST corner detector provided as a part of OpenCV library. The threshold for keypoint detection was dynamically adjusted using PID controller (to match the desired count of stereo matches).

On ASLAM Sensor platform, keypoint detection algorithm (Harris corner detector) is implemented in FPGA core, saving therefore the CPU time. Currently, the keypoints are detected using constant threshold and then filtered using their Harris response value:

Input : Detected Harris corners, desired keypoint count
Output: Most salient keypoints, no more than desired count
if *keypoints.count* > *desired_count* **then**
  sort *keypoints* according to Harris response, descending;
  truncate *keypoints* to *desired_count* number of elements;
end
return *keypoints*;

Descriptor extraction

Detected corner points will need to be matched in pairs between left-right frames and current-previous frames. One of the basic approaches is to compare image patches using some variant of cross-correlation function. This approach shows some deficiencies:

- comparing image patches is not rotationally invariant – may fail in case of highly dynamic scenes
- accessing image patches may be slow on embedded devices and requires to store image information for a longer time (till the next iteration)

The other approach is to use some specialised routines called descriptors that would extract valuable and unique information from the environment of the keypoint and store it in some minimalistic way. To match the keypoints on an embedded device, BRISK descriptor [10] is one of the best choices, as it provides easily-comparable binary output (size of 64 bytes = 512 bits), good performance and rotational-invariance. To obtain a dissimilarity measure of two binary descriptors, Hamming distance can be used – it will return the total number of bits that differ in 64-byte chunks.

What is even more, BRISK descriptor extraction will be soon implemented in the FPGA core of ASLAM Sensor, providing significant savings of CPU time.

Matching algorithm

Matching algorithm is used to match the keypoints between stereo frame pairs as well as between current and previous frames. The matching is done using BRISK descriptors and Hamming distance as dissimilarity measure (with highly optimized implementation, see 3.4.2). The fast comparison of descriptors permits to use brute-force matching with cross-checking.

Using some more sophisticated methods for matching algorithm, e.g. Flann matcher, may seem very tempting at first sight. The problem is that building the initial data structures (which will be done basically at every visual odometry iteration) is rather time-consuming and not worth the effort for small datasets [13].

Consistency checks

After matching the keypoints between pairs of left-right frames and current-previous frames, it is possible to introduce some simple and computationally-cheap methods to detect most evident matching mistakes.

For identified stereo correspondences, it is possible to perform epipolar geometry check. The epipolar constraint states that if coordinates of the point $\hat{\mathbf{x}}_l$ in the left frame are known, then it is possible to compute equation of the epipolar line $\mathbf{e}_r - \mathbf{x}_r$ in the right frame (assuming stereo rig extrinsics are known, $\mathbf{e}_r$ denotes right camera epipole). Having estimated the epipolar line parameters a, b, c, it is enough to check if $\hat{\mathbf{x}}_r$ lies on the line with some tolerance:

$$\text{dist}(ax + by + c = 0, \hat{\mathbf{x}}_r(u_r, v_r)) < \text{threshold} \tag{2.1}$$

$$\frac{|au_r + bv_r + c|}{\sqrt{a^2 + b^2}} < \text{threshold} \tag{2.2}$$

The estimation of the epipolar lines for a set of points is implemented in OpenCV library as `computeCorrespondEpilines` routine.

The epipolar geometry can only be used to check matching consistency between frames related by a known fundamental matrix. Unfortunately, when matching current frame points to previous frame points, there is no such estimate yet (the full pose is not estimated in prior update, see 2.3). The simplest approach would be to check if a point on the image plane "moved" between subsequent frames by no more than some time-dependent threshold:

$$\|\mathbf{x}_{\text{current}} - \mathbf{x}_{\text{previous}}\|_2 < \text{threshold} \cdot \Delta t \tag{2.3}$$

The aim of using consistency verification is to remove as many outliers as possible before starting RANSAC PnP procedure. Small number of outliers both increases the accuracy of visual odometry pose estimate as well as significantly speeds up RANSAC scheme (as it can quit before maximum number of iterations is reached, see 2.2.4 for RANSAC details).

2.2.3 Triangulation

One of the important parts of the algorithm pipeline is triangulation of stereo frame correspondences to get 3D coordinates of the keypoints that will be used in the next iteration to obtain pose update. Hartley's and Sturm's results in [14] suggest that commonly used linear triangulation with least squares minimization is in fact a good choice, as:

- It yields relatively accurate results
- It is fast and relies on linear algebra (which means it has potential to increase performance by using NEON engine, see 3.4.2)
- Can be easily extended to iterative LS method (see below)

Linear Least Squares triangulation

Linear triangulation relies on the fact that given corresponding points $\mathbf{u}_1$ and $\mathbf{u}_2$ (in homogeneous coordinates), as well as camera projection matrices $\mathbf{P}_1$ and $\mathbf{P}_2$ ($\mathbf{P}_i \in \mathbb{R}^{3\times4}$), we can relate them world coordinates $\mathbf{x}_w$ of the point by writing:

$$\mathbf{u}_i = \begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \mathbf{P}_i \cdot \mathbf{x}_w = \mathbf{P}_i \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{2.4}$$

We can decompose projection matrices into separate rows $\mathbf{p}_{i,1}$, $\mathbf{p}_{i,2}$ and $\mathbf{p}_{i,3}$, yielding:

$$wu = \mathbf{p}_{i,1}^T \cdot \mathbf{x}_w \tag{2.5}$$
$$wv = \mathbf{p}_{i,2}^T \cdot \mathbf{x}_w \tag{2.6}$$
$$w = \mathbf{p}_{i,3}^T \cdot \mathbf{x}_w \tag{2.7}$$

where w is some scale-factor. It can be eliminated by substituting $\mathbf{p}_{i,3}^T \cdot \mathbf{x}_w$:

$$u_i \mathbf{p}_{i,3}^T \cdot \mathbf{x}_w = \mathbf{p}_{i,1}^T \cdot \mathbf{x}_w \tag{2.8}$$
$$v_i \mathbf{p}_{i,3}^T \cdot \mathbf{x}_w = \mathbf{p}_{i,2}^T \cdot \mathbf{x}_w \tag{2.9}$$

We can now create a system of equations in form of $\mathbf{Ax} = 0$, with equations obtained from both stereo frames:

$$\begin{bmatrix} u_1\mathbf{p}_{1,3}^T - \mathbf{p}_{1,1}^T \\ v_1\mathbf{p}_{1,3}^T - \mathbf{p}_{1,1}^T \\ u_2\mathbf{p}_{2,3}^T - \mathbf{p}_{2,1}^T \\ v_2\mathbf{p}_{2,3}^T - \mathbf{p}_{2,1}^T \end{bmatrix} \mathbf{x}_w = 0 \tag{2.10}$$

Assuming $[x, y, z, 1]^T$ form of world coordinate points, we have 4 equations (2 from each frame) to estimate 3 unknowns. By further manipulations, we end up with non-homogeneous equation system of standard form:

$$\mathbf{A}\mathbf{x}_w = \mathbf{b} \tag{2.11}$$

The resulting non-homogeneous, overdetermined system can be solved by different methods, Hartley and Sturm suggest using pseudo-inverse or Singular Value Decomposition methods [14]. It is worth to note that OpenCV offers `solve` routine that can solve such problem using SVD.

Iterative Linear Least Squares triangulation

The most serious objection to linear least squares triangulation is the fact that we are minimizing $\|\mathbf{A}\mathbf{x}_w\|_2^2$ (see equation 2.10), which has no geometric meaning [14] – specifically, it is not reflecting squared reprojection errors in image planes.

The meaningful minimization goal can be obtained by multiplying each row of $\mathbf{A}$ matrix in equation 2.10 by a weight:

$$\frac{1}{w_i} = \frac{1}{\mathbf{p}_{i,3}^T \mathbf{x}_w} \tag{2.12}$$

After such weighting, first row of the matrix will become:

$$\left(u_1\mathbf{p}_{1,3}^T - \mathbf{p}_{1,1}^T\right) \cdot \frac{1}{\mathbf{p}_{1,3}^T \mathbf{x}_w} = u_1 - \frac{\mathbf{p}_{1,1}^T}{\mathbf{p}_{1,3}^T \mathbf{x}_w} \tag{2.13}$$

which is a formula for reprojection error for the first coordinate in the first frame. Similar weighting will yield similarly meaningful results for other rows. There is one problem though – it is not possible to put the unknown $\mathbf{x}_w$ into the $\mathbf{A}$ matrix.

To solve this issue, an iterative algorithm can be used that will work as follows:

Input: Point coordinates in left and right frame, projection matrices
obtain first guess of $\mathbf{x}_w$ from LS-linear triangulation;
set initial weight values w_1 and w_2 to 1;
for $i \leftarrow 1$ **to** n **do**
  calculate new weight for frame 1: $w_1 = \mathbf{p}_{1,3}^T \mathbf{x}_w$;
  calculate new weight for frame 2: $w_2 = \mathbf{p}_{2,3}^T \mathbf{x}_w$;
  if $|w_1 - w_{1,previous}| < \epsilon$ ***or*** $|w_2 - w_{2,previous}| < \epsilon$ **then**
    break;
  end
  build $\mathbf{A}$ and $\mathbf{b}$, using w_1 and w_2;
  solve $\mathbf{A}\mathbf{x}_w = \mathbf{b}$ using SVD;
end
return $\mathbf{x}_w$;

Both types of triangulation were implemented and tested – please refer to 3.4.3 for quality and performance comparison.

2.2.4 RANSAC Perspective n-Point algorithm

Typically, RANSAC scheme paired with some kind of PnP algorithm is used to estimate camera pose out of given set of object points (points in world coordinates), corresponding image points and camera calibration matrices. PnP algorithms search for a pose that minimizes total reprojection error, i.e. sum of squared distances between projections of object (world) points and observed image points.

RANSAC scheme, described in detail below, is used to make the function resistant to outliers. RANSAC and PnP are often used in visual odometry algorithms, either to get rid of outliers before running some more sophisticated methods (e.g. bundle adjustment over larger number of frames [4][15][16]) or directly to get a pose estimate.

In case of of visual odometry on ASLAM Sensor, computational power constraints resulted in dropping any multi-frame non-linear pose refinement techniques, leaving RANSAC PnP followed by just a final non-linear optimization as an output of visual odometry block.

EPnP

In the final version of the project, EPnP algorithm [17] was used instead of OpenCV's default iterative reprojection error minimization using Levenberg-Marquardt method.

The EPnP method operates in a non-iterative way and requires $n \geq 4$. The idea is to express n 3D points as a weighted sum of four virtual control points. As a result, the problem reduces to solving a constant number of quadratic expressions in $O(n)$ time, instead of $O(n^5)$ in case of classic approaches. This brings a significant performance gain.

For timing comparison between iterative optimization method and EPnP, please refer to 3.4.3.

RANSAC scheme

RANSAC (RAndom SAmple Consensus) is an algorithm that is meant to fit a model to an observed data set, possibly with high percentage of outliers. It is an iterative procedure where model is fitted to multiple subsets of input data and the best guess is selected.

The exact steps of the pose estimation RANSAC PnP algorithm are as follows:

Input : world points, image points, camera intrinsics, k, m_{min}, n
Output: pose estimate in form of $[R;t]$
$[R;t]_{best} = [I;0]$;
$m_{best} = 0$ (initialize "best number of inliers" count);
$inliers = \emptyset$;
for $i \leftarrow 1$ **to** k **do**
  $m_{actual} = 0$;
  randomly select n points from the dataset;
  use PnP to obtain a pose guess $[R;t]$ from previously selected points;
  for $j \leftarrow 1$ **to** N **do**
    reproject world point $\mathbf{p}_{i,world}$ as $\mathbf{p}_{i,reproject}$;
    if $\|\mathbf{p}_{i,reproject} - \mathbf{p}_{i,image}\|_2 < threshold$ **then**
      $m_{actual} = m_{actual} + 1$;
      $inliers = inliers \cup \mathbf{p}_i$
    end
  end
  if $m_{actual} > m_{best}$ **then**
    $m_{best} = m_{actual}$;
    $[R;t]_{best} = [R;t]$;
  end
  if $m_{actual} > m_{min}$ **then**
    break;
  end
end
minimize reprojection error on $inliers$ set to get a final $[R,t]_{best}$;
return $[R;t]_{best}$;

where:

- k – desired number of iterations
- m_{min} – stopping condition (minimum number of inliers to stop)
- n – number of parameters needed for PnP algorithm (e.g. $n = 3$ for P3P)

The desired number of iterations k can be, at least approximately, estimated using the following formula:

$$k = \frac{\log(1-p)}{\log(1-(1-\epsilon)^n)} \tag{2.14}$$

where:

- p – confidence level
- ϵ – outlier percentage in dataset
- n – minimum sample size

If we desire $p = 0.995$ and assume $\epsilon = 0.2$ (which rarely happens assuming data consistency checks presented in 2.2.2) and $n = 3$ for P3P, we now that about 13 iterations are needed. This values corresponds with empirical results of the framework.

2.3 Visual-inertial data fusion

2.3.1 IMU model

In the subsequent sections, a simple IMU model is used, based on the ones presented in [18] and [19]. The sensors measurements are assumed to be corrupted by continuous Gaussian processes, which fulfills Kalman Filtering assumptions and makes the EKF an optimal filter in least-squares sense.

Gyroscope

A gyroscope measures the angular velocity of the body. The relationship between the measurement and the true angular velocity ω can be modelled as:

$$\boldsymbol{\omega}_m = \boldsymbol{\omega} + \mathbf{b}_\omega + \mathbf{n}_\omega \tag{2.15}$$

The measurement is corrupted with two elements: gyro bias $\mathbf{b}_\omega$ and gyro noise $\mathbf{n}_\omega$.

Noise term $\mathbf{n}_\omega$ is assumed to be a white Gaussian noise, equal for all three axes, characterised by:

$$E\left[\mathbf{n}_\omega\right] = 0 \tag{2.16}$$

$$Cov\left[\mathbf{n}_\omega\right] = \mathbf{N}_\omega \cdot \delta(\tau) = \sigma_\omega^2 \cdot \mathbf{I}_{3\times 3} \cdot \delta(\tau) \tag{2.17}$$

The gyro bias is modeled as a random walk process governed by a white Gaussian noise:

$$\dot{\mathbf{b}}_\omega = \mathbf{n}_{b,\omega} \tag{2.18}$$

$$E\left[\mathbf{n}_{b,\omega}\right] = 0 \tag{2.19}$$

$$Cov\left[\mathbf{n}_{b,\omega}\right] = \mathbf{N}_{b,\omega} \cdot \delta(\tau) = \sigma_{b,\omega}^2 \cdot \mathbf{I}_{3\times 3} \cdot \delta(\tau) \tag{2.20}$$

Accelerometer

The accelerometer measures the accelerations acting on the body. The sensor model is simplified, as no bias is assumed. So the measurement depends on true body accelerations $\mathbf{a}$, gravity vector $\mathbf{g}$ and white gaussian noise $\mathbf{n}_a$:

$$\mathbf{a}_m = C^T\left(\mathbf{a}_m - \mathbf{n}_a\right) + \mathbf{g} \tag{2.21}$$

Transformation C^T relates body frame and global frame.

Noise term, similarly to gyroscope case, is modelled by a white Gaussian noise:

$$E\left[\mathbf{n}_a\right] = 0 \tag{2.22}$$

$$Cov\left[\mathbf{n}_a\right] = \mathbf{N}_a \cdot \delta(\tau) = \sigma_a^2 \cdot \mathbf{I}_{3\times 3} \cdot \delta(\tau) \tag{2.23}$$

2.3.2 SC-EKF filter

As already mentioned in section 2.1, the visual-inertial filter is based on SC-EKF in order to properly deal with relative visual odometry measurements. Due to computational constraints, the filtering is done for attitude only.

The state of the attitude filter is represented by a quaternion, which is both computationally tractable and first of all resitant to any singularities (contrary to many other approaches, e.g. Euler angles). Taking into consideration the IMU models presented above and SC-EKF requirements, we arrive at following state representation:

$$x = \begin{bmatrix} \mathbf{q} \\ \mathbf{b}_\omega \\ \mathbf{q}_c \end{bmatrix} \tag{2.24}$$

where $\mathbf{q}_c$ is a cloned state, used to fuse relative visual odometry pose updates.

Some attention needs to be paid to state error representation:

$$\delta x = \begin{bmatrix} \delta\mathbf{q} \\ \delta\mathbf{b}_\omega \\ \delta\mathbf{q}_c \end{bmatrix} \tag{2.25}$$

As stated in [18], a simplest approach representing the state error (and covariance matrix) in terms of complete state vector will result in singularity of covariance matrix and numerical issues. Lefferts *et al.* suggests a few possible solutions, one of them is "Body-fixed covariance representation".

In this approach, state error is represented not as an arithmetic difference between the true quaternion $\mathbf{q}$ and estimated quaternion $\hat{\mathbf{q}}$, but as an error quaternion $\delta\mathbf{q}$ which relates true value and estimated value by a quaternion multiplication:

$$\mathbf{q} = \hat{\mathbf{q}} \otimes \delta\mathbf{q} \tag{2.26}$$

With an assumption of small error rotation angles, an approximation can be done:

$$\delta\mathbf{q} = \begin{bmatrix} \mathbf{k}\sin(\delta\boldsymbol{\Theta}/2) \\ \cos(\delta\boldsymbol{\Theta}/2) \end{bmatrix} \approx \begin{bmatrix} \frac{1}{2}\delta\boldsymbol{\Theta} \\ 1 \end{bmatrix} \tag{2.27}$$

From the approximation it follows that each error quaternion can be approximated by 3-element small angle rotation vector, therefore we end up at 9×9 covariance matrix P (3 entries for error state quaternion, 3 entries for gyroscope bias error, 3 entries for error cloned state quaternion).

The following sections present prior update of the filter (based on the gyroscope measurements) and posterior update of the filter (using visual odometry and accelerometer measurements) in detail.

Notation

The subscripts denote:

- $_{k|k-1}$ – prior value at time k
- $_{k|k}$ – posterior value at time k

Skew symmetric matrix of 4-dimensional vector (e.g. quaternion) is constructed as:

$$\lfloor \mathbf{q}\times \rfloor = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \tag{2.28}$$

Matrix $\mathbf{\Omega}(\boldsymbol{\omega})$ of vector $\boldsymbol{\omega}$ is given by:

$$\mathbf{\Omega}(\omega) = \begin{bmatrix} -\lfloor \boldsymbol{\omega}\times \rfloor & \boldsymbol{\omega} \\ -\boldsymbol{\omega}^T & 0 \end{bmatrix} \tag{2.29}$$

Initialization

We set initial values of the state variables and covariance matrix:

$$\mathbf{q}_{0|0} = [0,0,0,1]^T \tag{2.30}$$
$$\mathbf{q}_{c,0|0} = [0,0,0,1]^T \tag{2.31}$$
$$\mathbf{b}_{\omega,0|0} = [0,0,0]^T \tag{2.32}$$
$$\mathbf{P}_{0|0} = \mathbf{0}_{9\times 9} \tag{2.33}$$

Prior update

State propagation procedure generally follows the approach presented in [20], but includes a few modifications resulting from Stochastic Cloning EKF application (state is augmented by cloned quaternion) as suggested in [12].

1. Propagate gyroscope bias:

$$\mathbf{b}_{k+1|k} = \mathbf{b}_{k|k} \tag{2.34}$$

2. Correct gyro measurement:

$$\hat{\boldsymbol{\omega}}_{m,k+1|k} = \boldsymbol{\omega}_{m,k+1|k} - \mathbf{b}_{k+1|k} \tag{2.35}$$

3. Propagate state using corrected gyro measurement and 1st order integrator. By using this type of integrator, we assume that angular velocity is piecewise constant between iterations and equal to the average of two subsequent measurements:

$$\bar{\boldsymbol{\omega}} = \frac{\hat{\boldsymbol{\omega}}_{m,k+1} + \hat{\boldsymbol{\omega}}_{m,k}}{2} \tag{2.36}$$

Then, after computations and using Taylor series expansion, the updated state can be obatined using:

$$\begin{aligned} \mathbf{q}_{k+1|k} = \Bigg(\exp\left(\frac{1}{2}\mathbf{\Omega}(\bar{\boldsymbol{\omega}})\Delta t\right) + \frac{1}{48}\Big(\mathbf{\Omega}\left(\boldsymbol{\omega}_{k+1|k}\right)\mathbf{\Omega}\left(\boldsymbol{\omega}_{k|k}\right) \\ -\mathbf{\Omega}\left(\boldsymbol{\omega}_{k|k}\right)\mathbf{\Omega}\left(\boldsymbol{\omega}_{k+1|k}\right)\Big)\Delta t^2\Bigg)\mathbf{q}_{k|k} \end{aligned} \tag{2.37}$$

4. We can use the form and notation used by Lefferts *et al.* for computing state transition matrix with one slight modification – we need to add additional $\mathbf{I}_{3\times 3}$

block for the cloned state (identity means cloned state remains unchanged in propagation procedure):

$$\mathbf{\Phi}(t+\Delta t,t)=\begin{bmatrix}\mathbf{\Theta} & \mathbf{\Psi} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{I}_{3\times3}\end{bmatrix}_{9\times9} \tag{2.38}$$

Where, assuming $|\boldsymbol{\omega}|\to 0$, we define $\mathbf{\Theta}$ and $\mathbf{\Psi}$ as:

$$\lim_{|\boldsymbol{\omega}|\to0}\mathbf{\Theta}=\mathbf{I}_{3\times3}-\Delta t\lfloor\hat{\boldsymbol{\omega}}\times\rfloor+\frac{\Delta t^2}{2}\lfloor\hat{\boldsymbol{\omega}}\times\rfloor^2 \tag{2.39}$$

$$\lim_{|\boldsymbol{\omega}|\to0}\mathbf{\Psi}=-\mathbf{I}_{3\times3}\Delta t+\frac{\Delta t^2}{2}\lfloor\hat{\boldsymbol{\omega}}\times\rfloor+\frac{\Delta t^3}{6}\lfloor\hat{\boldsymbol{\omega}}\times\rfloor^2 \tag{2.40}$$

5. The last element needed to update covariance matrix is a discrete time noise covariance matrix $\mathbf{Q}_d$. We know that continuous time covariance matrix is given by:

$$\mathbf{Q}_{cont}=\begin{bmatrix}\sigma_\omega^2\cdot\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \sigma_{b,\omega}^2\cdot\mathbf{I}_{3\times3} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3}\end{bmatrix}_{9\times9} \tag{2.41}$$

According to [18], the final discretized matrix will have the following structure:

$$\mathbf{Q}_d=\begin{bmatrix}\mathbf{Q}_{11} & \mathbf{Q}_{12} & \mathbf{0}_{3\times3}\\ \mathbf{Q}_{12}^T & \mathbf{Q}_{22} & \mathbf{0}_{3\times3}\\ \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3} & \mathbf{0}_{3\times3}\end{bmatrix}_{9\times9} \tag{2.42}$$

where, again assuming $|\boldsymbol{\omega}|\to 0$, we have:

$$\mathbf{Q}_{11}=\sigma_\omega^2\Delta t\cdot\mathbf{I}_{3\times3}+\sigma_{b,\omega}^2\left(\mathbf{I}_{3\times3}\frac{\Delta t^3}{3}+\frac{2\Delta t^5}{5!}\cdot\lfloor\boldsymbol{\omega}\times\rfloor^2\right) \tag{2.43}$$

$$\mathbf{Q}_{12}=-\sigma_{b,\omega}^2\left(\mathbf{I}_{3\times3}\frac{\Delta t^2}{2}-\frac{\Delta t^3}{3}\cdot\lfloor\boldsymbol{\omega}\times\rfloor+\frac{\Delta t^4}{4}\cdot\lfloor\boldsymbol{\omega}\times\rfloor^2\right) \tag{2.44}$$

$$\mathbf{Q}_{22}=\sigma_{b,\omega}^2\Delta t\cdot\mathbf{I}_{3\times3} \tag{2.45}$$

$$\tag{2.46}$$

6. Final step of state propagation is to update covariance matrix:

$$\mathbf{P}_{k+1|k}=\mathbf{\Phi}\mathbf{P}_{k|k,cloned}\mathbf{\Phi}^T+\mathbf{Q}_d \tag{2.47}$$

Posterior update

In the posterior update step, previously propagated state is corrected using measurements from visual odometry and accelerometer.

1. The first step is to obtain visual odometry residual – we just need to relate attitude change from visual odometry with attitude change between previously cloned state and current propagated state:

$$\delta\mathbf{q}_{res,VO}=\left(\mathbf{q}_c^{-1}\otimes\mathbf{q}_{k+1|k}\right)^{-1}\otimes\mathbf{q}_{VO} \tag{2.48}$$

and then get a 3-element small angle approximation of residual:

$$\delta\mathbf{q}_{res,VO}\approx\begin{bmatrix}\frac{1}{2}\delta\mathbf{\Theta}_{res,VO}\\ 1\end{bmatrix} \tag{2.49}$$

2. Obtaining residual quaternion based on accelerometer data is a bit more involved. The problem is, accelerometer used as gravity reference cannot be used to uniquely determine orientation (as the angle along the axis parallel to gravity vector is unknown). While the problem can be solved using Euler angles representation, this approach can cause problems with singularities (gimbal lock problem). On the other hand, a quaternion-based solution requires a full state estimate based on measurement.

   In [21], Madgwick *et al.* presented an idea of using a single gradient descent step, starting from the current state, which over few filter iterations will bring the state to the correct orientation with respect to gravity.

   The optimization task can be defined as :

$$\min_{\mathbf{q}} f(\mathbf{q}, \mathbf{d}, \mathbf{s}_a) \tag{2.50}$$

   where the function f is defined as:

$$f(\mathbf{q}, \mathbf{b}, \mathbf{s}_a) = \mathbf{q}^{-1} \otimes \mathbf{d} \otimes \mathbf{q} - \mathbf{s}_a \tag{2.51}$$

   So the optimization can be understood as looking for such orientation $\mathbf{q}$ that reference gravity vector $\mathbf{d}$ (we expect it to point downwards) in body coordinates agrees with normalized accelerometer measurement $\mathbf{s}_a$. A single gradient descent iteration (with step size μ) can be performed following:

$$\mathbf{q}_{gravity} = \mathbf{q}_{k+1|k} - \mu \frac{\nabla f(\mathbf{q}, \mathbf{b}, \mathbf{s}_a)}{\|\nabla f(\mathbf{q}, \mathbf{b}, \mathbf{s}_a)\|} \tag{2.52}$$

   and then can be feeded to equation below to get a measurement residual quaternion:

$$\delta \mathbf{q}_{res,gravity} = \left(\mathbf{q}_{k+1|k}\right)^{-1} \otimes \mathbf{q}_{gravity} \tag{2.53}$$

   and then get a 3-element small angle approximation of residual:

$$\delta \mathbf{q}_{res,gravity} = \begin{bmatrix} \frac{1}{2}\delta \mathbf{\Theta}_{res,gravity} \\ 1 \end{bmatrix} \tag{2.54}$$

3. Compute the measurement matrix $\mathbf{H}$ – it will be calculated based on linearized residual calculation. The matrix has the form:

$$\mathbf{H} = \begin{bmatrix} \mathbf{H}_{VO} \\ \mathbf{H}_{gravity} \end{bmatrix}_{6\times 9} = \begin{bmatrix} \left.\frac{\partial(\delta \mathbf{\Theta}_{res,VO})}{\partial(\delta x)}\right|_{x_{k+1|k}} \\ \left.\frac{\partial(\delta \mathbf{\Theta}_{res,VO})}{\partial(\delta x)}\right|_{x_{k+1|k}} \end{bmatrix}_{6\times 9} \tag{2.55}$$

4. Construct residual vector (residual will contain residuals from 2 sources – visual odometry and accelerometer used as gravity reference, both represented using small angle approximation):

$$\mathbf{r} = \begin{bmatrix} \delta \mathbf{\Theta}_{res,VO} \\ \delta \mathbf{\Theta}_{res,gravity} \end{bmatrix}_{6\times 1} \tag{2.56}$$

5. Compute measurement covariance matrix $\mathbf{R}$:

$$\mathbf{R} = \begin{bmatrix} \sqrt{\Delta t} \cdot \sigma_{VO}^2 \cdot \mathbf{I}_{3\times 3} & \mathbf{0}_{3\times 3} \\ \mathbf{0}_{3\times 3} & \sqrt{\Delta t} \cdot \sigma_a^2 \cdot \mathbf{I}_{3\times 3} \end{bmatrix} \tag{2.57}$$

6. Compute residual covariance matrix:

$$\mathbf{S} = \mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R} \tag{2.58}$$

7. Obtain Kalman gain matrix:

$$\mathbf{K} = \mathbf{P}\mathbf{H}^T\mathbf{S}^{-1} \tag{2.59}$$

8. Compute the correction:

$$\begin{bmatrix} \delta_{corr}\boldsymbol{\Theta} \\ \Delta_{corr}\mathbf{b} \\ \delta_{corr}\boldsymbol{\Theta}_c \end{bmatrix}_{9\times 1} = \begin{bmatrix} 2\cdot\delta_{corr}\mathbf{q} \\ \Delta_{corr}\mathbf{b} \\ 2\cdot\delta_{corr}\mathbf{q}_c \end{bmatrix} = \mathbf{K}\mathbf{r} \tag{2.60}$$

9. Update the state quaternion – note that $\delta_{corr}\mathbf{q}$ uses small angle representation so we need to convert it back to full quaternion form so that we can update the quaternion state. Conversion works as follows:

$$\delta\mathbf{q} = \begin{cases} \begin{bmatrix} \delta_{corr}\mathbf{q} \\ \sqrt{1 - \delta_{corr}\mathbf{q}^T\delta_{corr}\mathbf{q}} \end{bmatrix} & \text{if } x\|\delta_{corr}\mathbf{q}\|_2^2 \leq 1 \\ \\ \dfrac{1}{\sqrt{1 + \delta_{corr}\mathbf{q}^T\delta_{corr}\mathbf{q}}} \cdot \begin{bmatrix} \delta_{corr}\mathbf{q} \\ 1 \end{bmatrix} & \text{if } \|\delta_{corr}\mathbf{q}\|_2^2 > 1 \end{cases} \tag{2.61}$$

10. Eventually update the state:

$$\mathbf{q}_{k+1|k+1} = \delta\mathbf{q} \otimes \mathbf{q}_{k+1|k} \tag{2.62}$$

11. Update the bias:

$$\mathbf{b}_{k+1|k+1} = \mathbf{b}_{k+1|k} + \Delta_{corr}\mathbf{b} \tag{2.63}$$

12. Update covariance matrix:

$$\mathbf{P}_{k+1|k+1} = (\mathbf{I}_{9\times 9} - \mathbf{K}\mathbf{H})\,\mathbf{P}_{k+1|k}\,(\mathbf{I}_{9\times 9} - \mathbf{K}\mathbf{H})^T + \mathbf{K}\mathbf{R}\mathbf{K}^T \tag{2.64}$$

13. Clone the state:

$$\mathbf{q}_c = \mathbf{q}_{k+1|k+1} \tag{2.65}$$

14. Reflect the state cloning in covariance matrix, that means copy the state covariance matrix blocks to cloned state covariance blocks. Having the covariance matrix in a form:

$$\mathbf{P}_{k+1|k+1} = \begin{bmatrix} P_q & P_{q,b_\omega} & P_{q,q_c} \\ P_{q,b_\omega}^T & P_{b_\omega} & P_{b_\omega,q_c} \\ P_{q,q_c}^T & P_{b_\omega,q_c}^T & P_{q_c} \end{bmatrix}_{9\times 9} \tag{2.66}$$

we want to get:

$$\mathbf{P}_{k+1|k+1,cloned} = \begin{bmatrix} P_q & P_{q,b_\omega} & P_q \\ P_{q,b_\omega}^T & P_{b_\omega} & P_{b_\omega,q} \\ P_q^T & P_{b_\omega,q}^T & P_q \end{bmatrix}_{9\times 9} \tag{2.67}$$

Chapter 3

Implementation details

3.1 ASLAM Sensor hardware platform

ASLAM Sensor (see fig. 3.1) is an embedded Linux-based device using a Zynq-7000 family CPU from Xilinx. It is controlled by PetaLinux operating system that is provided by Xilinx as most convenient development environment for Zynq systems on a chip. The exact specifications are as follows:

- Zynq-7020 dual-core ARM 667 MHz with FPGA core
- Two Aptina MT9V034 cameras with baseline of 11 cm, providing 720 x 480 8-bit grayscale images at 30 fps
- Industrial-grade ADIS16448 IMU unit
- Gigabit Ethernet port, used for streaming data
- Size: 133 x 40 x 57 mm
- Weight: 130 g

As one may notice, the dual-core ARM running at rather limited speed can pose a relatively big challenge to run computer vision algorithms using it. On the other hand, an algorithm not requiring an external, powerful CPU will bring a lot of benefits – due to limited weight, size and power consumption ASLAM Sensor can be easily integrated into existing designs, such as UAVs.

3.2 Programming language and libraries

The implementation of the algorithm on the ASLAM Sensor was developed using C++ language (compliant to C++98) together with 3 libraries:

- OpenCV 2.4.7 – used for vision-related processing
- Eigen – used for Stochastic Cloning EKF implementation, it supported quite smooth migration from Matlab-prototyped code to C++
- Boost 1.53 – used for system-specific functionalities, such as TCP/IP support, threading, time measurements

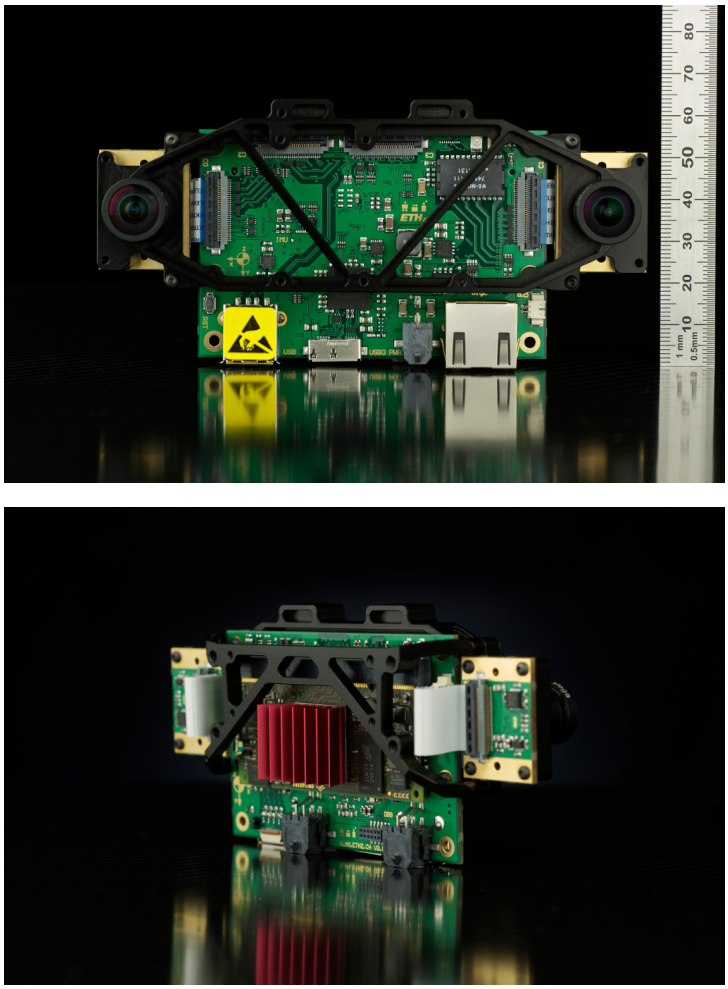


Figure 3.1: ASLAM Sensor: front and CPU side.

3.3 Sensor data interface

In the current ASLAM Sensor framework, sensor data (both for IMU and cameras) is gathered by FPGA and placed in ring-buffers. The framework provides user-friendly routines e.g. to get the pointer to image and IMU data, extract data timestamps or move the pointer to the next buffer element. Converting ASLAM Sensor databuffer to OpenCV `Mat`-type image is really simple as in both cases the pixels have depth of 8-bits and row-major format. It is enough to use just one line:

```
Mat frame(480, 752, CV_8UC1, (void*)(sensor.getCameraDataPointer(0) + 4));
```

The offset of 4 bytes is introduced as the first 32bits (unsigned long) of the image data are used by timestamp.

The images (and all the large data structures) are passed in the visual odometry framework by constant references to avoid making local copies of objects. It means that the algorithm uses only the image data stored in a ring buffer memory. A word of caution is necessary here, because we need some special care to ensure no data races can occur – the buffer size needs to be properly adjusted to expected iteration runtime.

3.4 Performance and optimizations

3.4.1 Initial sofware performance

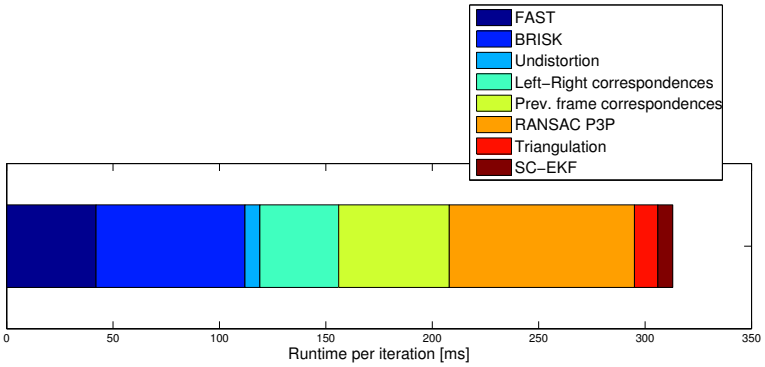


Figure 3.2: Initial performance of the algorithm on ASLAM Sensor.

Initial performane of the algorithm on the ASLAM Sensor was not very satisfying. The framework hardly reached the level of 3 Hz, which is a prohibitive value for the egomotion estimaton task. Certainly, optimization measures were necessary.

Figure 3.2 presents the initial (i.e. before optimizations) runtime of the algorithm, divided into most notable parts. Based on this timing results, some conclusions were made:

- RANSAC P3P procedure is very time consuming and number of iterations should be limited
- Descriptor matching takes over 100 ms in total, it is a surprisingly large value for simple binary descriptor matching using Hamming distance
- Keypoint detection and BRISK descriptor extraction should be parallelized and, if possible, outsourced to FPGA

3.4.2 ARM NEON optimizations

NEON overview

ARM NEON technology introduces some additonal ARM/Thumb functionalities, including specific instruction set and registers we can operate on. It is often called SIMD technology, as it introduces **S**ingle **I**nstruction **M**ultiple **D**ata philosophy.

The performance gain of SIMD approach comes from a fact that current applications (e.g. computer vision, algorithms, audio) often operate on data that is narrower than 32-bit bus width of ARM processor, e.g. 8-bit (e.g. single byte pixel values of grayscale image) or 16-bit variables. As a consequence, in each instruction, only a limited part of the CPU capabilities is used.

SIMD instructions address this inefficiency – we can perform the same instruction on a couple of data quants in parallel. Let's imagine we want to sum two 4D vectors with values in range of $0 \div 255$ (single byte integers). Normally, we would need 4 addition instructions, but only one NEON pairwise sum is enough. That gives 4-fold performance boost in this case.

It is very convenient for developers that ARM NEON instructions can be used inside ARM/Thumb assembly code [22]. The code can be compiled together with standard source code, execution can be controlled by standard ARM instructions and finally it is possible to debug it.

ARM NEON code can be written in two formats:

- **NEON C intrinsics** – by including `arm_neon.h` header file, it is possible to directly use ARM assembly instructions that will operate on existing C/C++ variables
  e.g. `temp = vmlal_u8 (temp,rgb.val[1], gfac);`
- **NEON assembly** – where we directly write assembly code and need to take care about copying data between general purpose registers and NEON registers on our own
  e.g. `vmull.u8 q8, d16, d22`
  It is worth to note that currently[1] NEON assembly methods provides significantly better performance than C intrinsics.

Specific NEON registers

ARM NEON operations are executed on NEON registers, which consist of 32 64-bit (quad-word) registers (D0 till D31). The registers can be also addressed as 16 128-bit registers, i.e. Q0-Q15. Some care needs to be taken when choosing the registers we operate on, as D8-D15 are callee-saved and we should restore their values when exiting the procedure (which takes time).

When using NEON assembly code, we need to manually copy data between general purpose registers (that means, our C/C++ variables) and NEON registers. Let's have a look at an example where we multiply operate on single-precision floating point numbers. Note how `vldmia` and `vstmia` instructions are used to copy data:

```
// load data from general purpose registers %1 and %2
// into NEON registers
"vldmia %1, { q8-q11 } \n\t"
"vldmia %2, { q1 } \n\t"

// do some data processing
"vmul.f32 q0, q8, d2[0]\n\t"
"vmla.f32 q0, q9, d2[1]\n\t"
"vmla.f32 q0, q10, d3[0]\n\t"
"vmla.f32 q0, q11, d3[1]\n\t"

// write output to general purpose register %0
"vstmia %0, { q0 }"
```

Hamming distance using assembly NEON

As already mentioned in section 3.4.1, BRISK descriptor matching time was significantly longer than expected. It may seem that a simple Hamming norm distance (for each pair of 64-byte descriptors, a number of differing bits is calculated) should be really fast, but $O(n^2)$ OpenCV procedure provided very unsatisfactory results (e.g. 28 ms for 358 keypoints). After some research, it turned out that NEON engine offers instructions perfectly matching Hamming distance calculation, namely:

- `VEOR` – calculating *exclusive or* over 128-bit vectors (NEON Q-registers)

[1] We should expect that gcc/g++ compilers will gradually get better at compiling C intrinsics NEON code and nearly catch up assembly code performance

- `VCNT` – calculating set bit count over 128-bit vectors

A full NEON assembly code was written, providing about 45% performance boost compared to original OpenCV function (see figure 3.3).

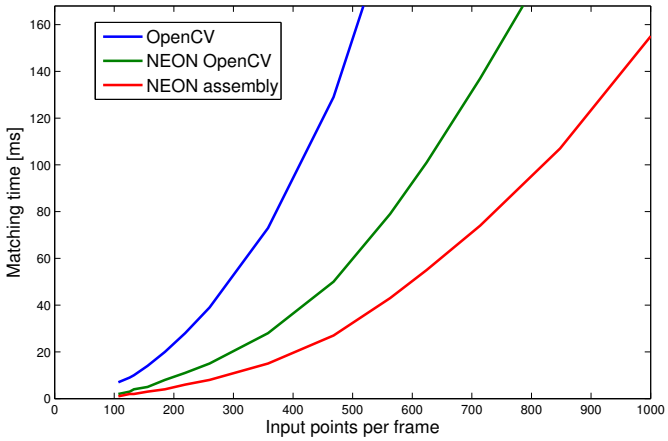


Figure 3.3: Comparison of Hamming matching times of descriptors for about 200 keypoints.

The Hamming distance function `NEON_batchDistHammingCV` is implemented as a method of `NeonMath` class. It has an identical interface to the original OpenCV routine (but does not support masked input and assumes 64-byte descriptors).

Other NEON functionalities

Similarly, some other parts of the algorithm could be easily rewritten using ARM NEON assembly language. Although they were not as critical as descriptor matching phase, these optimizations still provided a total gain of over 8 milliseconds. `NeonMath::NEON_MultiplyPoint2f` routine implements multiplication of 2D point (expressed in camera coordinates) by camera intrinsic matrix.

3.4.3 Visual odometry optimizations

RANSAC PnP timing

Initially, RANSAC PnP took a very long time and what is even worse, the duration was highly varying depending on number of iterations needed. The first step in the optimization process was to determine, what exactly is that computationally expensive in the RANSAC PnP routine.

After investigation, it turned out it is the iterative PnP algorithm that takes about 95% of the total RANSAC PnP time. The decision was to keep parts that reproject points and count inliers not modified and have a closer look at possible PnP optimizations.

The switch to EPnP method caused the ration to significantly decrease, but still average out at about 76% of total RANSAC PnP time. The histogram showing the distribution of this ratio for EPnP is depicted in the figure 3.4.

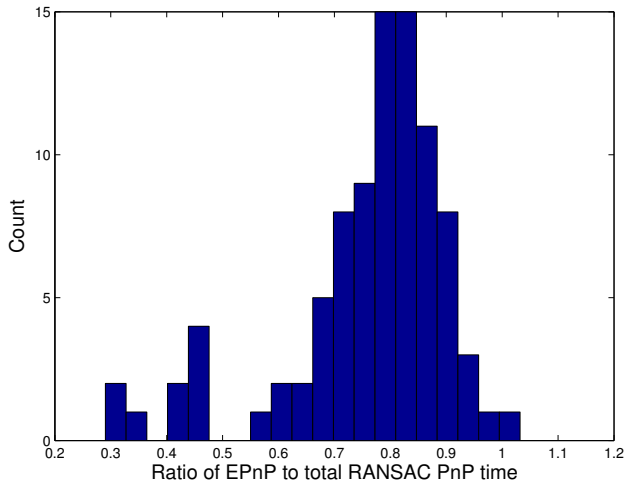


Figure 3.4: Ratio of EPnP time to total RANSAC PnP time, average of about 76%.

PnP initial condition

As expected, providing a reasonable initialization can significantly improve the performance of iterative PnP methods (e.g. when using `CV_ITERATIVE` flag in `SolvePnP` OpenCV function). The simplest assumption for initialization is $[R;t] = [I;0]$, i.e. no pose change. Visual odometry operates at relatively high sampling rate so this value is not very far from truth.

Such change indeed siginificantly speeded up RANSAC PnP procedure (see 3.5), but also introduced big delays once per 10-20 frames (delay of about 400 ms). Therefore, the approach was dropped and other methods were used.

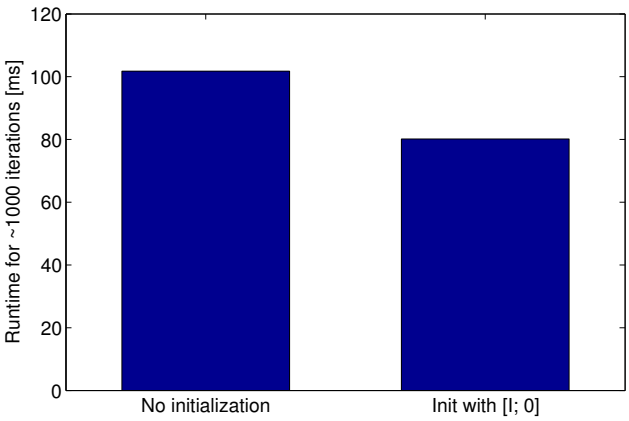


Figure 3.5: Comparison of runtime of OpenCV iterative PnP method when not initialised and initialised with $[R;t] = [I;0]$.

PnP method

One of the most significant time-savings comes from a switch from `CV_ITERATIVE` method to `CV_EPNP` method that uses EPnP method to compute pose update (see 2.2.4). While EPnP may give slightly worse results than iterative minimization of reprojection error, it is very reliable when used in RANSAC scheme. As a consequence, the total runtime of RANSAC PnP method dropped from about 80 ms to about 20 ms (see 3.6).

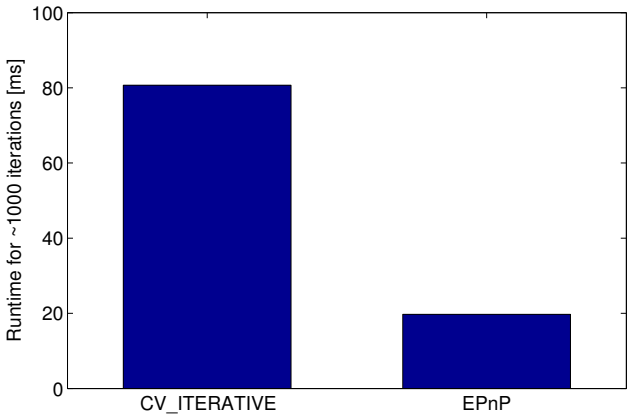


Figure 3.6: Comparison of runtime of PnP methods: iterative method vs EPnP.

Data quality vs RANSAC-EPnP runtime

One of the questions that was asked when discussing the design of the algorithm is why to use strict epipolar geometry or motion constraints (and invest CPU time in them) if RANSAC procedure should take care of removing outliers on its own.

While this is of course possible, it must be noted that better quality data makes the RANSAC procedure come up with a good guess much quicker and possibly quit thanks to the stopping condition. To verify this hypothesis, a dataset was collected where a number of total RANSAC EPnP iterations (with stopping condition equal to 95% of input points) was related to the value of epipolar constraint check.

As expected and clearly visible in the figure 3.7, the number of iterations increases when the threshold is relaxed. Taking into account that each EPnP iteration takes about 1.2 to 2 milliseconds, the savings when using data consistency checks are significant (consistency checks take around 2 ms in total).

Triangulation

Both linear-LS and iterative linear-LS triangulation methods were implemented on ASLAM Sensor. Theoretically, iterative version of triangulation routine should improve reconstruction results and as a consequence pose estimation output.

In practice, iterative triangulation only slightly changed the results of the egomotion estimation framework, in order of milimeters after hundreds of frames. Taking into account current accuracy, the additional CPU time spent on iterative procedure (comparison presented in the figure 3.8) brings no advantages. The final decision was to use standard linear-LS triangulation.

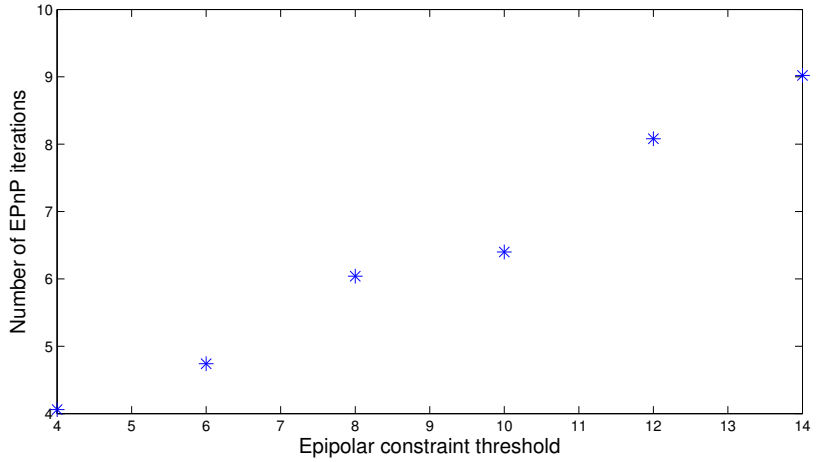


Figure 3.7: Average number of iterations of EPnP depending on epipolar constraint threshold.

3.4.4 Parallelization

ASLAM Sensor is equipped with a dual-core ARM processor and therefore it is possible to run two threads in parallel. The visual odometry pipeline consists of block that can be easily parallelized, as:

- image/keypoint/descriptor preprocessing is done separately for each of the stereo frames
- keypoints between current and previous frames are matched independently
- triangulation operates on each pair of left-right image points separately
- RANSAC together with PnP algorithm works iteratively and iterations do not depend on each other

All above mentioned visual odometry blocks were parallelized (figure 3.9), bringing a significant gain in total iteration runtime.

Implementation of the parallel routines was implemented manually, using Boost library. More automated approaches to parallelization, like Threading Building Blocks library or OpenMP library are very effective in multi-core high-performance computing, but introduce quite heavy overhead. An example of such manual parallelization code looks as follows:

```
boost::thread thread1(boost::bind(&VisualOdometry::threadFunction, this,
    boost::ref(arg1), boost::ref(arg2), ...));
boost::thread thread2(boost::bind(&VisualOdometry::threadFunction, this,
    boost::ref(arg1), boost::ref(arg2), ...));
thread1.join();
thread2.join();
```

Please note that the larger data structures should be passed by reference (in this case Boost reference) to avoid creating local copies of objects in thread function.

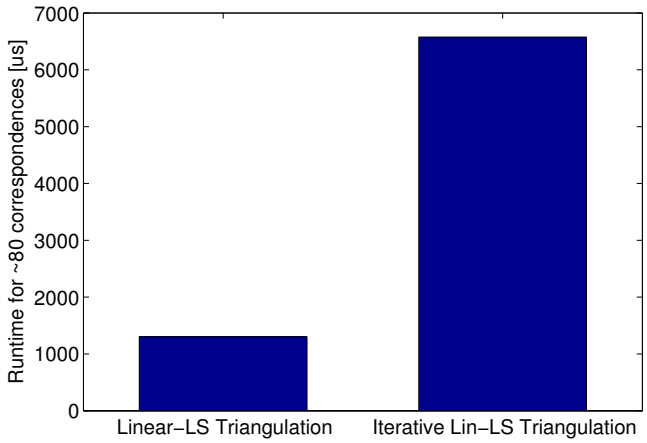


Figure 3.8: Comparison of runtime of triangulation routine for linear least squares triangulation and iterative linear triangulation (for 80 stereo matches). The ratio between two values may tell us how many iterations are needed for iterative method to converge.

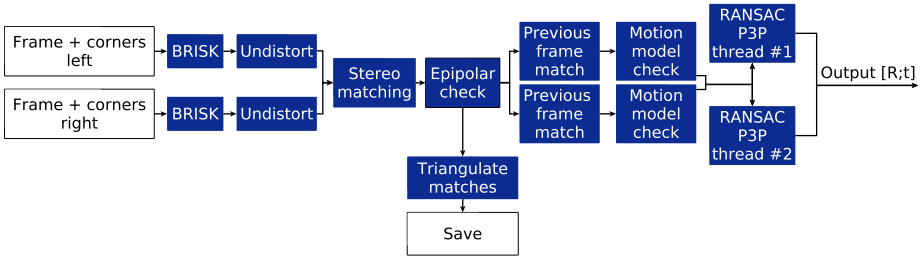


Figure 3.9: Detailed view at visual-odometry pipeline, including parallelization.

3.4.5 CPU-specific compiler optimizations

One of the most obvious ways to speed up execution is to use optimization flags when compiling the source code.

- `-O3` – maximum possible degree of optimization
- `-march=armv7-a` – specifying ARM7 architecture as target
- `-mtune=cortex-a9` – some further optimizations to ARM Cortex-A9 CPUs
- `-ffast-math` – speeds up math operations, but also introduces some possible unsafe optimizations (e.g. only finite values of variables, some roundings, lack of ISO standards conformity)
- `-ftree-vectorize` – try to vectorize loops
- `-fomit-frame-pointer` – procedures will not have to save, set and restore frame pointer if it is not needed for them
- `-mfpu=neon` – enabling NEON optimizations

- `-mfloat-abi=softfp` – option required to use NEON engine, please note that the code will still use full hardware floating-point instructions (option `-mfloat-abi=soft` would make all floating point calculation in software)

3.5 Parametrization

3.5.1 Visual odometry

Most often tweaked parameters of the visual odometry algorithm are included in the calibration file and can be modified without a need to recompile the source code:

- `desiredFeatureCount` (default: `300`) – defined in calibration file, desired number of keypoints that are used in the algorithm, values between 200 and 400 provide good results and reasonable quality
- `ransacMaxIters` (default: `150`) – defined in calibration file, maximum number of iterations of RANSAC procedure if stopping condition was not fulfilled earlier; setting it too low will dramatically deteriorate pose estimation quality; setting it too high may introduce peaks in the algorithm runtime for frames with large count of outliers
- `ransacThreshold` (default: `3.0`) – defined in calibration file, threshold for RANSAC PnP procedure that is used to check if a point is an inlier, constant expressed in pixels
- `ransacMinInliers` (default: `0.94`) – defined in calibration file, stopping criterion for RANSAC PnP algorithm (minimum number of inliers should be: `ransacMinInliers*numberOfPoints`); if set too low, it will negatively affect performance of estimation; if set too high, RANSAC will need more iterations and algorithm runtime will increase
- `minShutterWidth` (default: `25`) – defined in calibration file, minimum coarse shutter width of MT9V034; useful for high-contrast outdoor scenes where camera automatic exposure settings are adjusted e.g. to sky brightness and close objects are too dark
- `maxShutterWidth` (default: `150`) – defined in calibration file, maximum coarse shutter width of MT9V034; useful for high-contrast scenes where the camera adjust the shutter width to a value that seems to be too large

Some less frequently changing parameters are hard-coded as constants in `VisualOdometry` class and need recompilation to be modified:

- `initialSoftwareFastThreshold` (default: `12`) – defined in code, initial FAST corner detector threshold, used only one FPGA Harris corner detector fails to provide corners
- `stereoEpipolarThreshold` (default: `4.0`) – defined in code, maximum deviation from the epipolar line to treat a pair of stereo matches as correct
- `featureDisplacementThreshold` (default: `250`) – defined in code, maximum displacement of a keypoint between consecutive frames; if it is larger than this value then the keypoint is rejected

- `minP3pPoints` (default: `10`) – defined in code, the minimum number of input points to PnP algorithm to return a value, otherwise zero pose change is assumed

- `maxDistancePerIteration` (default: `0.8`) – defined in code, maximum distance traveled in each iteration of visual odometry, expressed in meters (if visual odometry return translation larger than this value that it is ignored and zero-translation is assumed); this parameter is used as a robustness-increasing measure, but in fact reaching the threshold never happended during testing phase

- `TIMING` – defined in code, if the constant is defined, visual odometry timing information is displayed (warning: a lot of output is produced using this option, it can affect the overall performance of the algorithm)

3.5.2 SC-EKF filter

Stochastic cloning EKF filter is used to fuse visual and inertial attitude information. The only parameters that are needed for it's operation are standard deviations of the measurement sources (including gyroscope bias standard deviation used to model bias evolution).

- `sigmaGyroNoise` (default value: `1e-7`) – defined in code, standard deviation of gyroscope noise

- `sigmaGyroBias` (default value: `4*1e-6`) – defined in code, standard deviation of gyroscope bias

- `sigmaVisualOdo` (default value: `5*1e-6`) – defined in code, standard deviation of visual odometry rotation output

- `sigmaAccelNoise` (default value: `1e-4`) – defined in code, standard deviation of accelerometer noise

3.5.3 Other parameters

- `tcpPort` (default value: `1234`) – defined in calibration file, TCP/IP port for connection with PC client (see 3.6)

Some debugging parameters are also set in `main.cpp` file, they can come helpful when verifying different functionalities of the ASLAM sensor, FPGA output and the algorithm itself:

- `USE_TEST_FRAMES` – defined in code, use offline frames only, no image is read from the sensor

- `CHECK_TIMING` – defined in code, use `TimingTest` class and offline image files to run timing tests (current implementation tests matching speed only)

- `CHECK_KEYPOINTS` – defined in code, read frames and corners from FPGA, overlay corners on the corresponding images and save them as JPG files

3.6 ROS PC-side client

Output of the algorithm on the ASLAM sensor can be streamed over TCP/IP, together with one of the camera frames. Sensor advertises TCP/IP service on user-selected port (default: 1234). A dedicated PC-side application, compiled as ROS package, can connect to ASLAM sensor and publish the received data as ROS messages. The following topics are advertised:

- `/aslam_sensor/odometry` – algorithm output published as ROS odometry message that contains position and orientation data (orientation is expressed as quaternion)
- `/aslam_sensor/cam0` – standard ROS image format with image stream from camera0 of the ASLAM sensor, useful mainly for visualizations; note that new image is published on each algorithm iteration (so currently at about 10 Hz)

The data can be visualized using RViz, a ROS 3D visualization tool.

Chapter 4

Results

4.1 Pose estimation results

First stage of testing online performance of the egomotion estimation algorithm was done in CLA building. Most of the datasets were closed trajectories, to check what will be the final deviation from origin (e.g. figures 4.2b and 4.1).

Some other aspects were tested on the CLA staircase. The dataset was collected when walking up the stairs from ground floor up to the top floor. Heavy rotation and change of vertical position (floors have similar heights) was properly reflected in the dataset. The resulting trajectory is depicted in the figure 4.2a.

One outdoor dataset (trajectory depicted in 4.1) also tested resistance of the algorithm to dynamic lighting changes (when entering and leaving the bulding) as well as ability to reject moving objects (pedestrians and cars appear in the dataset). It is possible to conclude that RANSAC did a good job with outlier rejection as no clear disturbances of trajectory are visible.

One of the questions that needed to be answered was how the Stochastic Cloning EKF improves the results of pure Visual Odometry. It turned out that especially the accelerometer used as a gravity reference brings a significant improvement, as we gain an absolute measurement for pitch and roll axis. In the figure 4.3, it is clearly visible how accelerometer used in posterior update improved the reliability of vertical axis position.

To further inspect the performance, a comparison was made with Stefan Leutenegger's visual-inertial SLAM approach [2] that can be considered as pseudo-ground truth information for the lightweight approach we are dealing with. The comparison of a trajectory in CLA building of ETH is depicted in the figure 4.4.

4.2 Final performance on ASLAM sensor

The final performance of the algorithm on the ASLAM Sensor is satisfactory and reaches levels of about 15 Hz rate, depending on the exact parameters chosen and whether the image streaming to a PC is turned on. The optimization attempts (see 3) brought a vast improvement of the algorithm performance. Greatest savings were made thanks to:

- outsourcing keypoint detection to the FPGA core

- parallelization
- migration to EPnP for Perspective n-Point method
- NEON optimizations

The final breakdown by algorithm stages is presented in the figure 4.5.

The distribution of iteration times was also investigated and can be seen in the figure 4.6, observed over 1500 iterations. The data was collected during motion in keypoint rich environment. It is possible to draw a few conclusions of the histogram plot: it is notable that almost one third of the iterations fall below 50 ms limit and almost all fit within 100 ms limit (i.e. 10 fps). There are absolutely no samples exceeding 200ms (in fact, maximal runtime was 182 ms), which is rather a good information about performance stability of the algorithm.

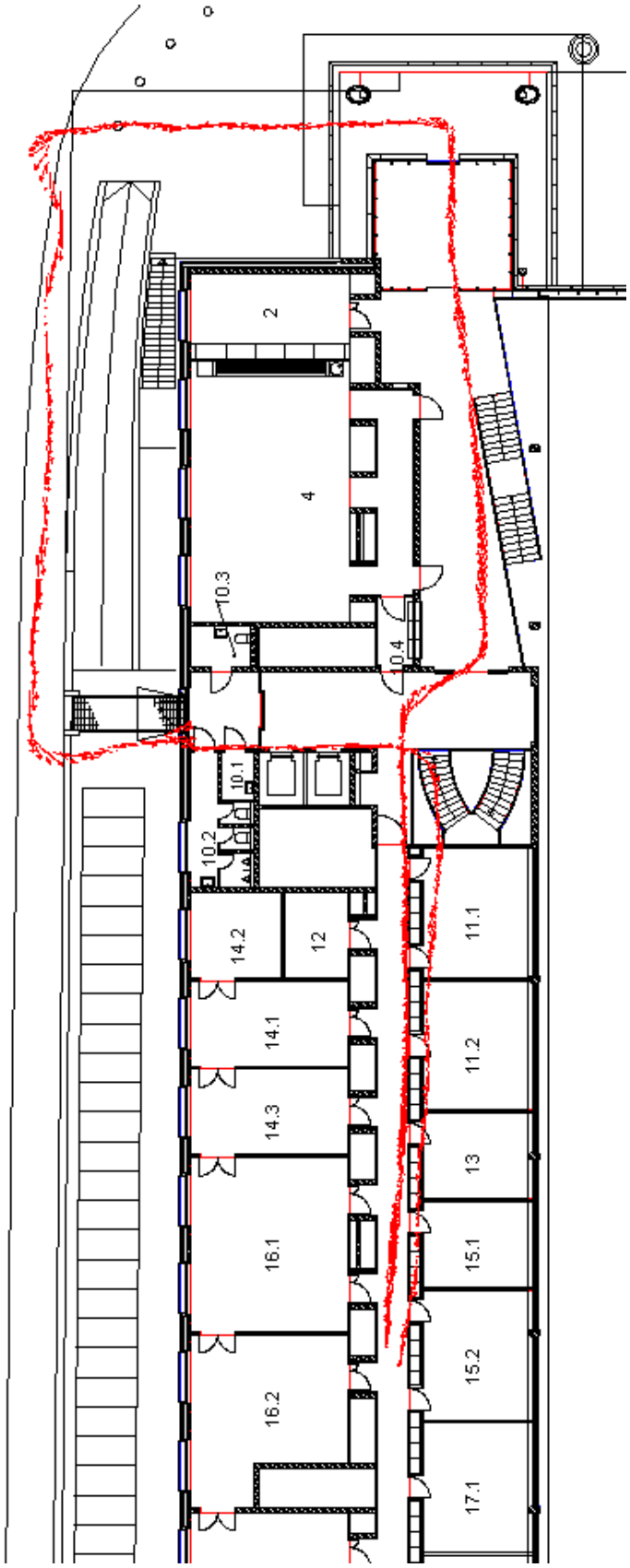
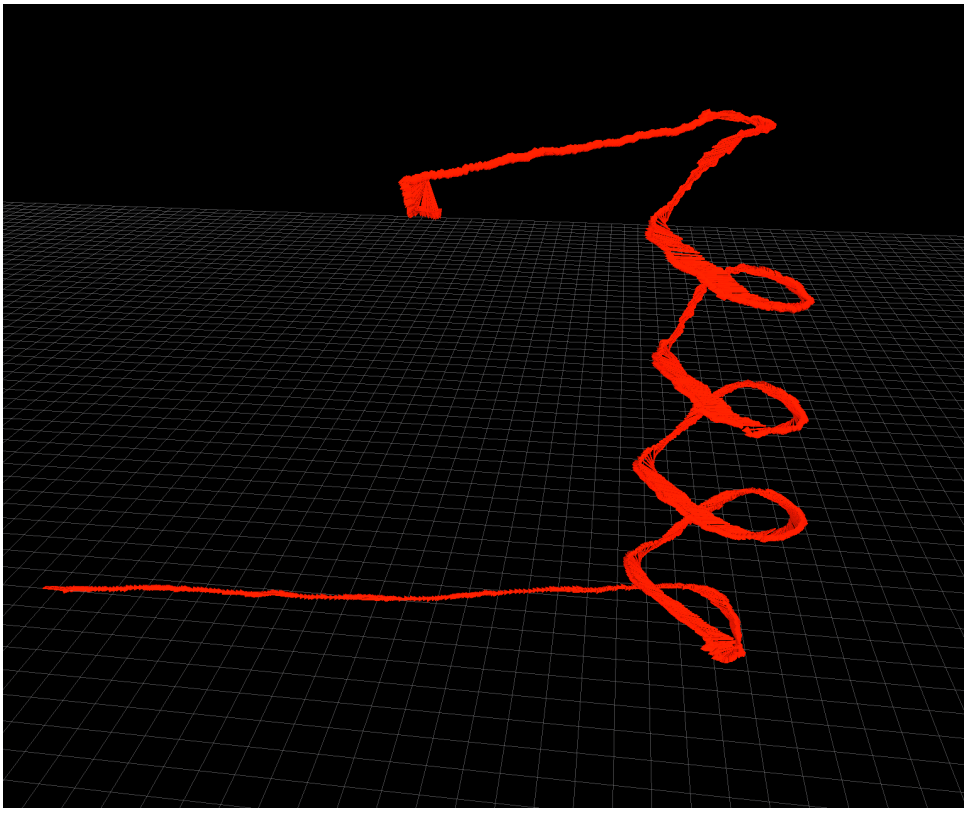
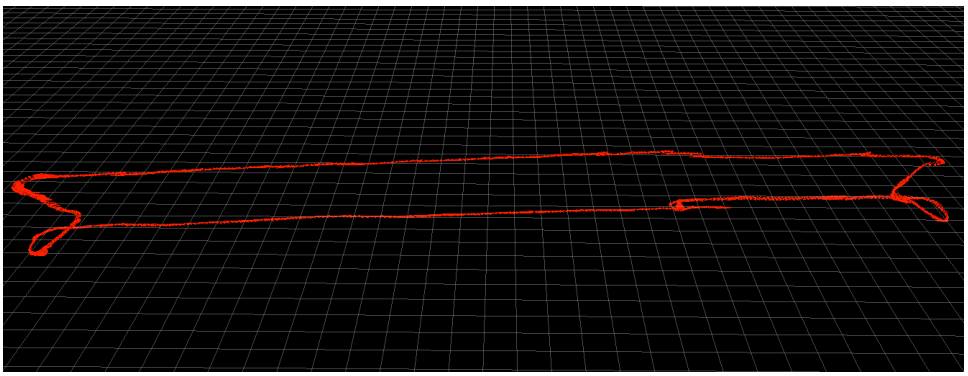


Figure 4.1: A dataset overlayed on the CLA building plan. Final position error was equal to about 2.8 m.

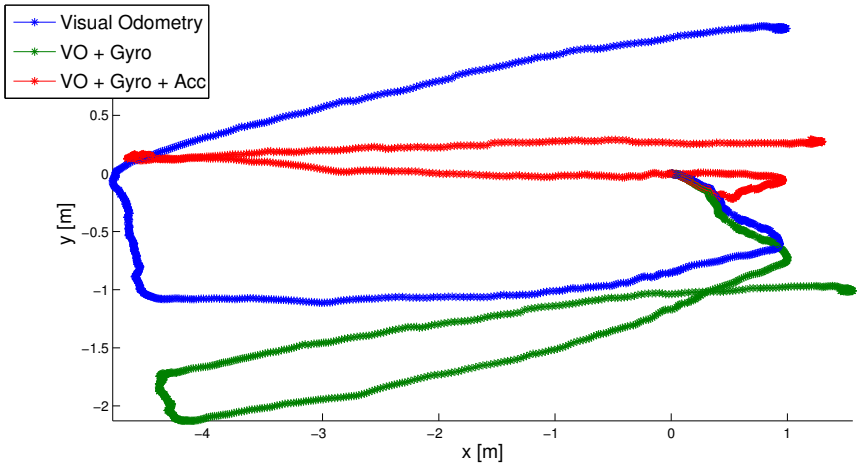


(a) CLA staircase from E floor till J floor



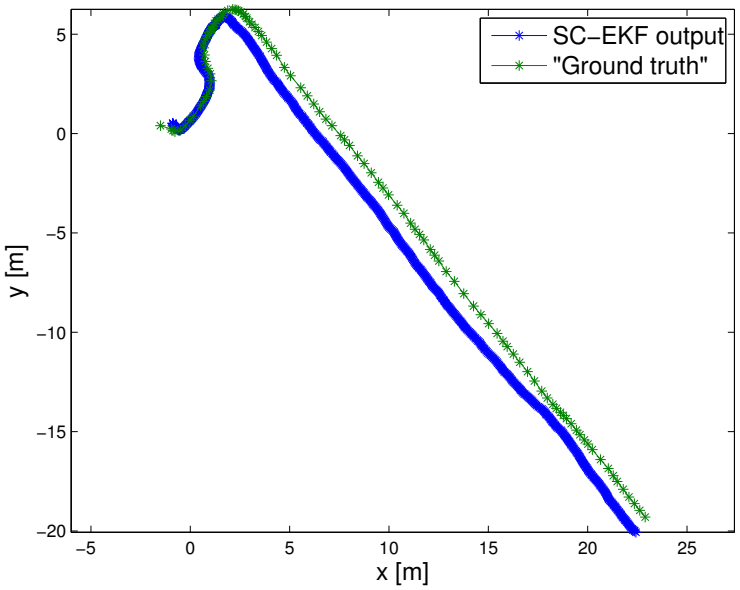
(b) Checking of output consistency and loop closure error on 2 floors of CLA building. The final error was equal to about 3.45 m deviation from initial position.

Figure 4.2: Evaluation of the online egomotion estimation in ETH CLA building



[h]

Figure 4.3: Side view of the flat trajectory. Comparison between unfiltered visual odometry (blue), no accelerometer SC-EKF (green) and SC-EKF using accelerometer as gravity reference (red).



[h]

Figure 4.4: Comparison of the algorithm performance (blue curve) with S. Leutenegger's visual-inertial SLAM [2]. RMS error of ...

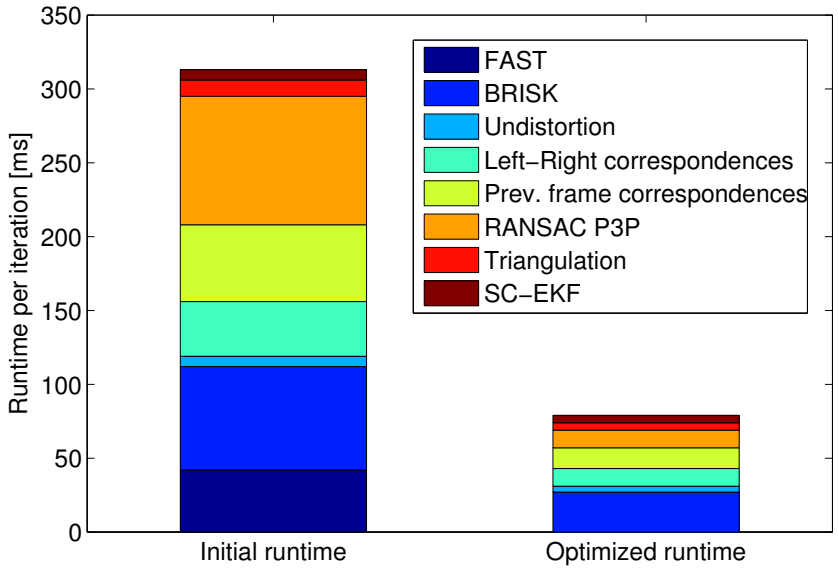


Figure 4.5: Comparison between initial and final performance of the algorithm on ASLAM Sensor.

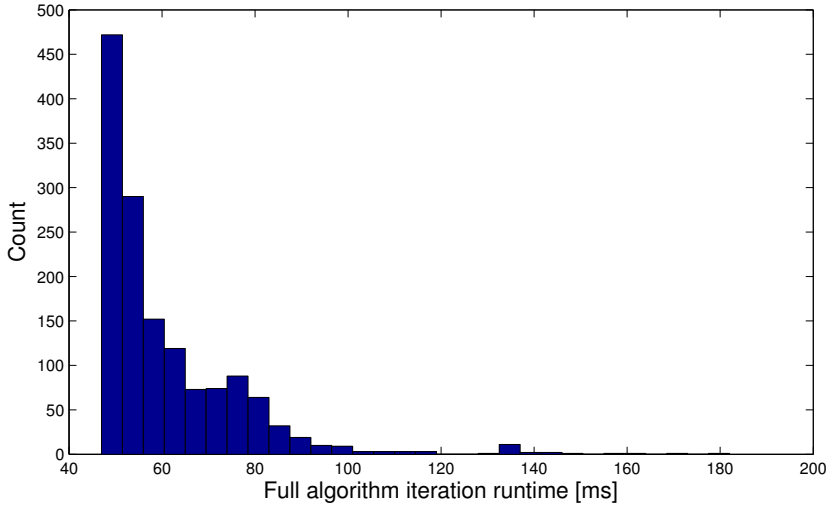


Figure 4.6: Histogram of total algorithm runtime per iteration, without streaming images over Ethernet. Mean iteration time: 61 ms.

Chapter 5

Conclusion and outlook

In this project, a concept of lightweight egomotion estimation algorithm using visual-inertial measurements has been derived and implemented. The algorithm was tested both on offline datasets as well as during online evaluations on ASLAM Sensor.

The visual odometry algorithm is based upon a simple frame-to-frame matching using BRISK binary descriptors, that guarantee high reliability. Binary descriptors are relatively easy to compute and can be matched efficiently using specific ARM NEON instructions. The pose estimates are obtained using RANSAC scheme and EPnP [17] algorithm, that proved to be most effective in the current implementation. The visual odometry pipeline is highly parallelized, which provides additional performance boost on a dual-core ARM CPU.

The visual-inertial data fusion is realised using Stochastic Cloning EKF. While the filter remains computationally cheap, it properly treats dependent relative pose updates provided by visual odometry algorithm. Because of computational constraints, the filter is used to filter attitude only, leaving translation unfiltered. Using the filter significantly improved pose estimation quality, especially accelerometer measurements used as gravity reference helped to converge to true orientation in pitch and roll axis, as they are used as an absolute measurement. The final errors of the framework, over trajectories of hundreds of meters, usually remain in range of single meters.

The third work package was to optimize the algorithm, so that it could be used online on ASLAM Sensor. A few different techniques were used to obtain better performance, including parallelization, using more efficient algorithms and support for ARM NEON engine. The average single iteration runtime dropped from 320 ms to about 75 ms. The variance of iteration runtimes is rather low, they rarely exceed 100 ms and maximum over 1500 test frames equals to 182 ms.

Naturally, there are a few potential directions where the project should be developed further to achieve better results and/or performance. Certainly, some other parts of visual odometry pipeline (e.g. BRISK descriptor extraction, PnP) can be outsourced to FPGA, saving over 50% of the current runtime so that the algorithm would reach 20 FPS level. A significant quality gain is expected from introducing full-state SC-EKF that will filter not only the attitude but also the position. Adding magnetometer measurements in such filter would help to converge to true yaw angle. Furthermore, temporal drift of the position can be avoided by multi-frame visual odometry and (heuristic) management of triangulated keypoints.

Appendix A

Usage

A.1 Installing the framework on ASLAM Sensor

A few steps are needed to prepare the sensor for the use with egomotion estimation framework:

1. SSH to the sensor: `ssh root@10.0.0.1`
2. Remount the filesystem in read-write mode: `mount -o remount,rw /`
3. Copy new FPGA bitsream together with loader script into `/home/root/fpga` directory, you can use Nautilus for that (File → Connect to server...)
4. Copy OpenCV library files to `/usr/lib` directory
5. Copy executable file `egomotion.elf` to root home directory `/home/root/`
6. Copy calibration file `calib.yaml` to root home directory `/home/root/`
7. Reboot the sensor: `reboot`, the new FPGA bitstream should be loaded now

A.2 Starting the egomotion estimation framework

Starting egomotion estimation code on ASLAM Sensor is really simple, as all the parameters are predefined in calibration/settings file. It is enough to call:

```
./egomotion.elf
```

Note that current FPGA/CPU interface together with device drivers are not always working properly when started the first time. You may need to restart the framework to detect IMU.

A.3 Compiling the PC client

ASLAM Sensor PC client is delivered as catkin ROS package. Therefore it can be compiled similar to a standard ROS package. The dependencies include OpenCV and Boost. To get the package compiled, follow the steps:

1. Put `aslam_sensor_client` folder in `catkin_ws/src` directory (by default, `catkin_ws` will be located in your home directory)
2. Go one directory up: `cd ..`
3. Compile your catkin packages: `catkin_make`
4. Observe the output. The package should compile without any errors and warnings.

A.4 Starting the PC client

When the framework on ASLAM Sensor is already started, it is also accepting TCP/IP connections. Then you can start your PC side client by issuing:

```
rosrun aslam_sensor_client client 10.0.0.1
```

Note that the last argument is an IP address of the sensor that may change depending on sensor configuration.

Appendix B

Calibration file structure

The calibration file that is used by the egomotion estimation framework must conform to the OpenCV YAML reader/writer format. The following variables should be initialized in the calibration file:

- `minShutterWidth`
- `maxShutterWidth`
- `tcpPort`
- `desiredFeatureCount`
- `ransacMaxIters`
- `ransacThreshold`
- `ransacMinInliers`
- `Rcc` – camera-to-camera rotation matrix (3×3 matrix)
- `Tcc` – camera-to-camera translation vector (3×1 vector)
- `Rbc` – IMU-to-camera rotation matrix (3×3 matrix)
- `f0` – focal length of camera 0 (2×1 vector)
- `d0` – distortion parameters of camera 0 (4×1 vector, equidistant model)
- `p0` – principal point of camera 0 (2×1 vector)
- `f1` – focal length of camera 1 (2×1 vector)
- `d1` – distortion parameters of camera 1 (4×1 vector, equidistant model)
- `p1` – principal point of camera 1 (2×1 vector)

For parameter descriptions refer to section 3.5. Example calibration files can be found on project github page[1], in `calibration_files` directory.

[1] `https://github.com/ethz-asl/aslam_visual_inertial_egomotion`

Appendix C

Files

Git repository

The directory structure on Git repository[1]:

- `aslam_sensor_client` – PC-side ASLAM Sensor client that can publish ROS messages (camera 0 frame + pose estimate)
- `calibration_files` – examples of calibration files that are used by the ego-motion estimation framework
- `egomotion` – ASLAM Sensor code, stored as Xilinx SDK 14.7 project (note that it uses relative symlinks and you need to checkout `aslam_drivers` in parent directory)
- `test_frames` – files that can be put on sensor for testing purposes (test frames + calibration files)

DVD disc

The directory structure on CD attached to paper version of the report:

- `bags` – ROS rosbags with algorithm evaluation datasets
- `code` – sofware, organised as in Git repository
- `presentation` – final presentation of the semester thesis
- `report` – this report
- `sensor` – OpenCV libraries to be put in `/usr/lib` and FPGA bitstream
- `video` – final version of YouTube video

[1] `https://github.com/ethz-asl/aslam_visual_inertial_egomotion`

Bibliography

[1] L. Kneip, M. Chli, and R. Siegwart, "Robust real-time visual odometry using a single camera and an IMU," in *Proceedings of the British Machine Vision Conference (BMVC)*, 2011.

[2] S. Leutenegger, P. Furgale, V. Rabaud, M. Chli, K. Konolige, and R. Siegwart, "Keyframe-based visual-inertial slam using nonlinear optimization," in *Proceedings of Robotics: Science and Systems*, (Berlin, Germany), June 2013.

[3] K. Schmid and H. Hirschmuller, "Stereo vision and imu based real-time egomotion and depth image computation on a handheld device," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pp. 4671–4678, May 2013.

[4] R. Voigt, J. Nikolic, C. Huerzeler, S. Weiss, L. Kneip, and R. Siegwart, "Robust embedded egomotion estimation," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pp. 2694–2699, Sept 2011.

[5] H. Hirschmuller, P. Innocent, and J. Garibaldi, "Fast, unconstrained camera motion estimation from stereo without tracking and robust statistics," in *Control, Automation, Robotics and Vision, 2002. ICARCV 2002. 7th International Conference on*, vol. 2, pp. 1099–1104 vol.2, Dec 2002.

[6] D. Nister, O. Naroditsky, and J. Bergen, "Visual odometry," in *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, vol. 1, pp. I–652–I–659 Vol.1, June 2004.

[7] A. Howard, "Real-time stereo visual odometry for autonomous ground vehicles," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 3946–3952, Sept 2008.

[8] D. Scaramuzza and F. Fraundorfer, "Visual odometry [tutorial]," *Robotics Automation Magazine, IEEE*, vol. 18, pp. 80–92, Dec 2011.

[9] S. Goldberg and L. Matthies, "Stereo and imu assisted visual odometry on an omap3530 for small robots," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pp. 169–176, June 2011.

[10] S. Leutenegger, M. Chli, and R. Siegwart, "BRISK: Binary Robust Invariant Scalable Keypoints," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2011.

[11] A. I. Mourikis, S. I. Roumeliotis, and J. W. Burdick, "SC-KF mobile robot localization: A Stochastic Cloning-Kalman filter for processing relative-state measurements," 2006.

[12] S. Roumeliotis and J. Burdick, "Stochastic cloning: a generalized framework for processing relative state measurements," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 2, pp. 1788–1795 vol.2, 2002.

[13] M. Muja and D. G. Lowe, "Fast matching of binary features," in *Computer and Robot Vision (CRV)*, pp. 404–410, 2012.

[14] R. Hartley and P. Sturm, "Triangulation," in *Computer Analysis of Images and Patterns* (V. Hlaváč and R. Šára, eds.), vol. 970 of *Lecture Notes in Computer Science*, pp. 190–197, Springer Berlin Heidelberg, 1995.

[15] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second ed., 2004.

[16] M. A. Lourakis and A. Argyros, "SBA: A Software Package for Generic Sparse Bundle Adjustment," *ACM Trans. Math. Software*, vol. 36, no. 1, pp. 1–30, 2009.

[17] V. Lepetit, F.Moreno-Noguer, and P.Fua, "EPnP: An accurate O(n) solution to the PnP problem," *International Journal Computer Vision*, vol. 81, no. 2, 2009.

[18] E. J. Lefferts, F. L. Markley, and M. D. Shuster, "Kalman filtering for spacecraft attitude estimation," *Journal of Guidance, Control, and Dynamics*, vol. 5, no. 5, pp. 417–429, 1982.

[19] S. Omari and G. Ducard, "Metric visual-inertial navigation system using single optical flow feature," in *Control Conference (ECC), 2013 European*, pp. 1310–1316, July 2013.

[20] N. Trawny and S. I. Roumeliotis, "Indirect Kalman filter for 3D attitude estimation," Tech. Rep. 2005-002, University of Minnesota, Dept. of Comp. Sci. & Eng., Mar. 2005.

[21] S. Madgwick, R. Vaidyanathan, and A. Harrison, "An efficient orientation filter for inertial measurement units (IMUs) and magnetic angular rate and gravity (MARG) sensor arrays," tech. rep., Department of Mechanical Engineering, University of Bristol, April 2010.

[22] ARM Ltd., *NEON architecture overview*, 2013.