Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

An Optical Flow Odometry Sensor Based on the Raspberry Pi Computer

Adam Heinrich

Supervisor: Dr. Gaël Écorchard
Field of study: Open Informatics
Subfield: Computer Vision and Image Processing
January 2017

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student:	Bc. Adam H e i n r i c h
Study programme:	Open Informatics
Specialisation:	Computer Vision and Image Processing
Title of Diploma Thesis:	An Optical Flow Odometry Sensor Based on the Raspberry Pi Computer

Guidelines:

The Raspberry Pi single-board computer embeds a graphics processing unit that includes a motion estimation block normally used to evaluate frame changes during hardware video encoding. The vectors from this motion estimation block can be obtained directly from the GPU and provide a hardware estimation of the image displacement. By combining this information with a distance and orientation sensor one can estimate the robot's velocity, thus its position. The aim of the work is to implement an odometry sensor based on this solution, in the way of a PX4Flow sensor.

Student's tasks:
1. Bibliographic research about optical flow, presentation of existing solutions.
2. Implementation of a mixed GPU/CPU solution for pose estimation.
3. Study of the possibility of an implementation of a pure CPU solution for pose estimation.
4. Experimental verification.

Bibliography/Sources:

[1] Dominik Honegger, Lorenz Meier, Petri Tanskanen and Marc Pollefeys. An Open Source and Open Hardware Embedded Metric Optical Flow CMOS Camera for Indoor and Outdoor Applications, ICRA2013
[2] Itseez. The OpenCV Reference Manual (3.1), December 2015
[3] Richard Hartley and Andrew Zisserman. Multiple View Geometry in Computer Vision (Second Edition), Cambridge University Press, March 2014
[4] Iain E. G. Richardson. H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia, John Willey & Sons, 2003

Diploma Thesis Supervisor: Dr. Gaël Pierre Marie Ecorchard

Valid until: the end of the winter semester of academic year 2017/2018

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, May 26, 2016

Acknowledgements

I would like to thank Dr. Gaël Écorchard for his guidance and assistance with my thesis. I also thank Ing. Jan Chudoba for his assistence with outdoor experiments.

Finally, I would like to thank my parents for support and my girlfriend for patience.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 9 January 2017

Abstract

The thesis describes the design and implementation of an odometry sensor suitable for micro aerial vehicles.

The sensor is based on a ground-facing camera and a single-board Linux-based embedded computer with a multimedia SoC. The SoC features a hardware video encoder which is used to estimate optical flow in a real-time. The optical flow is then used in combination with a distance sensor to estimate vehicle's velocity.

The proposed sensor is compared to a similar existing solution and evaluated in both indoor and outdoor environments. Moreover, alternative software approaches, independent of the selected board's specific hardware and firmware implementation, are also proposed.

Keywords: visual odometry, optical flow, ego-motion

Supervisor: Dr. Gaël Écorchard

Abstrakt

Diplomová práce popisuje návrh a implementaci odometrického senzoru vhodného pro malé bezpilotní létající prostředky.

Senzor je založen na jednoteskovém počítači s operačním systémem Linux a kameře směřující k zemi. Počítač obsahuje hardwarový grafický čip, který během kódování videa počítá optický tok. Optický tok je spolu s informací se senzoru vzdálenosti použit pro odhad aktuální rychlosti pohybu.

Senzor byl porovnán s existujícím řešením a otestován v místnosti i ve venkovním prostředí. Práce také navrhuje alternativní softwarová řešení, která nejsou pevně svázána se specifickou hardwarovou implementací počítače.

Klíčová slova: vizuální odometrie, optický tok, ego-motion

Překlad názvu: Odometrický senzor na principu optical flow založený na počítači Raspberry Pi

Contents

Chapter 1
Introduction

The ability to estimate velocity is the fundamental task for the control of micro aerial vehicles (MAVs).

The aim of this work is to design and implement an odometry sensor similar to an existing solution. The existing sensor uses a ground-facing camera, ultrasonic distance sensor and a microcontroller to compute an optical flow and estimate the vehicle's velocity. Although the sensor's software is open-sourced, the microcontroller does not supposedly have enough resources for additional computer vision tasks. Also, the skills required to develop software for a highly constrained embedded system make the entrance barrier higher for potential developers.

The proposed solution is based on a widely available single-board computer which features a SoC designed for multimedia applications. The optical flow is computed by a hardware block rather than by the processor which leaves resources for additional tasks. The solution does not only measure the translational velocity but also measures the orientation, which is an improvement when compared to the existing sensor. Furthermore, purely software approaches which are not so closely coupled to the actual hardware are proposed.

The document is structured as follows: Chapter 2 provides an overview of the currently existing solutions. Chapter 3 summarizes the theoretical background. The design of the proposed solution is described in chapter 4. Chapters 5 and 6 focus on the experimental setup and an actual implementation. The implementation is then evaluated in chapter 7 and compared to the existing solution as well as to ground truth measurements. Finally, chapter 8 concludes the work and summarizes results.

Chapter 2
Related work

This chapter describes related work mainly in the field of embedded camera-based sensors for ego-motion estimation using optical flow.

2.1 PX4Flow

The PX4Flow [HMTP13] is an optical flow sensor based on a microcontroller with an ARM Cortex-M4 CPU. It has gained a significant popularity in hobby community. The software and schematic are open sourced.

Unlike other solutions, the PX4Flow uses an image sensor MT9V034 designed for machine vision with an electronic global shutter. The global shutter contributes to more reliable results as all pixels are exposed at the same time. The image acquisition system uses 4×4 pixel binning which improves its sensitivity in poor lighting conditions.

The flow is computed between two consecutive frames using a block-matching algorithm. The ARM Cortex-M4 CPU provides a set of special SIMD instructions which makes the block-matching process more efficient. The optical flow is then used for velocity calculation: in both horizontal and vertical directions a histogram of optical flow values is computed and the most frequent value is taken as the resulting velocity. The resulting velocity is then compensated for the effects caused by pitch and roll measurements from onboard MEMS gyroscope and converted to real-world units using the ultrasonic distance sensor.

The bare-metal architecture (without any operating system nor task pre-emption) allows fast processing: The flow is computed at 400 Hz. This high rate allows the CPU to search matching blocks in a small range of ± 4 pixels. The sensor uses a MavLink protocol to communicate with the host controller. It reports the current velocity in meters per second and the distance to the ground. Changes in MAV heading (yaw) are not measured and must be supplied by an external sensor (e.g. a digital compass) in order to integrate the measured velocity into position.

2.2 Solutions based on an optical mouse sensor

Optical flow sensors based on chips used in optical computer mice used to be quite popular especially in hobby community, probably due to good availability of these sensors and their low cost.

The most popular is the ADNS family by Avago Technologies, e.g. the ADNS-3080 [Ava08] chip with a SPI serial interface. Although it is possible to capture raw images, most applications use the internal digital signal processor to compute relative displacement between consecutive frames (this functionality is fundamental for the optical mouse operation).

It is usually possible to load a custom program to an internal ROM, the architecture of the DSP is however not disclosed. The chip provides resolution of 30×30 pixels and a frame rate from 2000 to 6469 FPS.

One of the available solutions is a part of the ArduPilot system [Ard16]. It is basically a breakout PCB for the ADNS-3080 sensor and a lens. The software uses the relative displacement values computed by the DSP. Integration of the sensor into a MAV relies on external sensors for ground distance and angular motion (to compensate for pitch and roll). Rotation around the center of the sensor (yaw) can not be recovered and is said to confuse the sensor.

Application described in [BZF13] uses inertial sensors and five ADNS-9500 optical mouse sensors aiming at different directions to implement visual odometry for MAV hovering. The authors introduce a translational optic-flow direction constraint (TOFDC) which only depends on optical flow direction and ignores its scale. This is important because it removes the dependency on an additional sensor (as the scale depends on a distance from the sensed environment) and it relaxes assumptions about environment's geometry (e.g. its flatness). The TOFDC constraint is then used to correct drifts of inertial sensors.

The dual-sensor approach described in [KB17] uses two ADNS-2051 optical mouse sensors and an inertial measurement unit for angular correction. The difference in two computed optical flows can be used to estimate the depth (and thus the velocity in real-world units), which removes the need for a separate distance sensor.

2.3 Other solutions

ArduEye is an embedded vision sensor based on the Stonyman ASIC by Centeye. The chip's low resolution allows image processing on a highly constrained embedded platforms such as the 8-bit Atmel AVR. The software is open-sourced and features multiple algorithms for optical flow computation, described in [SCN13]. The sensor seems to be already discontinued at the time of writing.

A visual odometry method described in [KG11] is also based on the ground-facing camera but does not use the optical flow for the motion estimation.

Instead the approach is based on a Fourier-Mellin transform which recovers both rotation and translation between two consecutive images.

A common approach which enables the usage of otherwise CPU-intensive algorithms in real-time systems is their implementation in FPGA, such as in the case of [KNP+12]. The application uses a side-looking camera and a FPGA implementation of the SURF feature detector. The displacement of features between consecutive frames is then used to estimate the MAV's altitude and yaw.

Chapter 3

Theory

This chapter describes the theoretical background of the proposed solution.

3.1 Optical flow

Optical flow is a displacement of pixel values in the image sequence induced by a movement of a camera or a scene observed by it. Let $I(u, v, t)$ be an image function of the pixel position (u, v) and time t. The optical flow between two frames captured at times t and $t + \Delta t$ can then be represented by the displacement $(\Delta u, \Delta v)$ and time difference Δt.

An example optical flow visualized using vector field is depicted on figure 3.1.

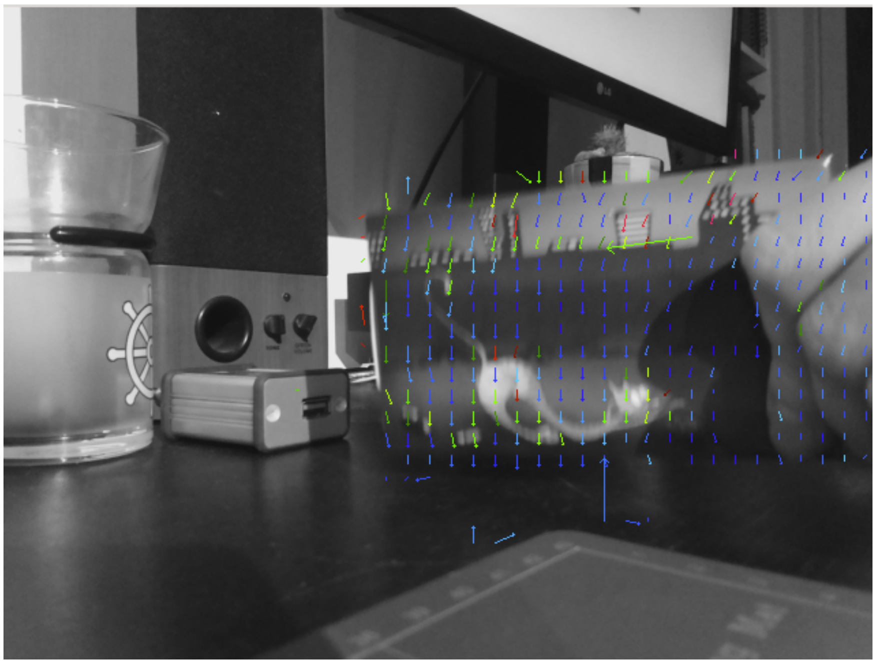

Figure 3.1: Optical flow induced by a object moving downwards

3.1.1 Constraint equations

Most approaches to optical flow estimation are based on a **brightness constancy constraint** (equation 3.1). This constraint assumes that moving pixels keep the same brightness between consecutive frames:

$$I(u,v,t) = I(u+\Delta u, v+\Delta v, t+\Delta t) \tag{3.1}$$

The brightness constancy can be linearized [WC11] using the Taylor approximation, which yields in

$$0 = I_u V_u + I_v V_v + I_t, \tag{3.2}$$

where I_u, I_v and I_t are partial derivates of the image function with respect to u, v and t, respectively and V_u and V_v are the velocities of the optical flow,

$$V = \begin{bmatrix} V_u & V_v \end{bmatrix} = \begin{bmatrix} \frac{\Delta u}{\Delta t} & \frac{\Delta v}{\Delta t} \end{bmatrix}. \tag{3.3}$$

As the equation 3.2 has two variables, it has an infinite number of solutions (which is known as the aperture problem). This ambiguity means that another constraints have to be enforced, such as the **spatial smoothness constraint**. The spatial smoothness constraint assumes that neighboring pixels belong to the same objects and therefore represent the same motion.

3.1.2 Optical flow algorithms

Lucas-Lanade algorithm

The Lucas-Kanade algorithm [LK+81] assumes a constant optical flow in small neighborhood of every pixel (u, v). The optical flow constraint (equation 3.2) is then applied to all pixels within the given window W, which results in a over-determined sets of equations. The flow is then estimated by minimizing sum of deviations using the least squares method:

$$\min_{\Delta u, \Delta v} \sum_{\boldsymbol{p} \in W(u,v)} [I_u(\boldsymbol{p})\Delta u + I_v(\boldsymbol{p})\Delta v + I_t(\boldsymbol{p})]^2 \tag{3.4}$$

The Lucas-Kanade algorithm estimates flow for a given set of pixels (ideally corners and textured patches detected by a feature tracker), its result is therefore a sparse optical flow. The method is valid only for small displacements.

Horn-Schmuck algorithm

The Horn-Schmuck algorithm [HS81] assumes global smoothness of the optical flow field to solve the aperture problem. The algorithm therefore minimizes derivatives of the optical flow field:

$$\min_{V_u, V_v} \iint [(I_u V_u + I_v V_v + I_t)^2 + \lambda(\|\nabla V_u\|^2 + \|\nabla V_v\|^2)] \mathrm{d}u \mathrm{d}v, \tag{3.5}$$

where $\boldsymbol{V} = [V_u(u,v), V_v(u,v)]^T$ is a displacement vector (optical flow) for an image pixel (u,v) and λ is a parameter to balance effects of both constraints. As the algorithm estimates flow for every pixel in an image, its result is a dense optical flow.

Block matching algorithm

The block matching algorithm is one of the simplest methods to compute the optical flow. For every pixel (u,v) in the original image, the closest match $(u+\Delta u, v+\Delta v)$ in the subsequent image is found by minimizing the Sum of Absolute Differences (SAD). The SAD value is computed by comparing a small (usually square) window $(M \times N)$ around the pixel (equation 3.6).

$$SAD = \sum_{i=-\frac{M}{2}}^{\frac{M}{2}} \sum_{j=-\frac{N}{2}}^{\frac{N}{2}} |I(u+\Delta u+i, v+\Delta v+j, t+\Delta t) - I(u+i, v+j, t)| \tag{3.6}$$

This method can be made faster by computing the flow only for a subset of image pixels instead of the full image matrix, producing only a sparse optical flow. It can be well parallelized as the flow can be computed independently for each pixel.

3.2 Ground-facing camera

A point in space $\boldsymbol{P} = [X, Y, Z]^T$ is projected by a pinhole camera (figure 3.2). The corresponding point on image plane $\boldsymbol{p} = [x, y, f]^T$ can be computed as

$$\boldsymbol{p} = -\frac{f}{Z}\boldsymbol{P}, \tag{3.7}$$

where f, the distance between the image plane and projection origin $\boldsymbol{O}$, is the camera's focal length. Since the camera is mounted perpendicularly to a vehicle body, the coordinate Z is equal to the distance between ground and camera's projection origin.

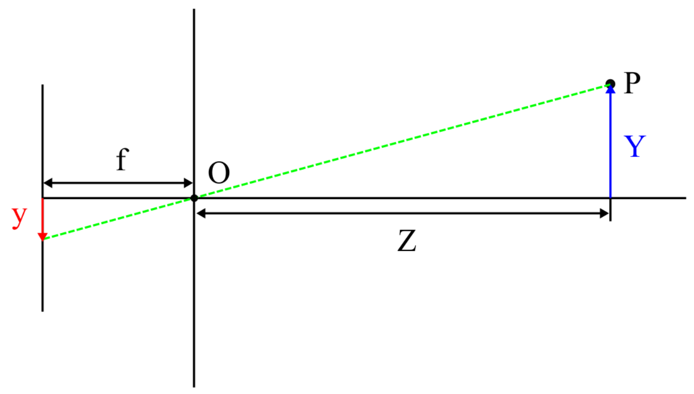

Figure 3.2: Pinhole camera model

The ground distance Z must be obtained from an external sensor, such as an ultrasonic or laser distance sensor or a barometric pressure sensor. Given the ground distance Z is approximately constant between two consecutive frames, a displacement in the image plane $(\Delta x, \Delta y)$ can be converted to a real word displacement $(\Delta X, \Delta Y)$:

$$\Delta X = -\frac{1}{f}\Delta x \cdot Z, \ \Delta Y = -\frac{1}{f}\Delta y \cdot Z. \tag{3.8}$$

The displacement in the image plane can be obtained using one of the Optical flow algorithms described in section 3.1. As the computed displacement $(\Delta u, \Delta v)$ is usually in pixels, it is required to convert it into real-world units (e.g. meters). Equation 3.8 then changes to

$$\Delta X = -\frac{s}{f}\Delta u \cdot Z, \ \Delta Y = -\frac{s}{f}\Delta v \cdot Z, \tag{3.9}$$

where s is the pixel size.

3.2.1 Angular correction

While the UAV is usually regulated to fly in a pose parallel to the ground, it is necessary to compensate small rotations between consecutive frames which manifest as an optical flow in the image plane.

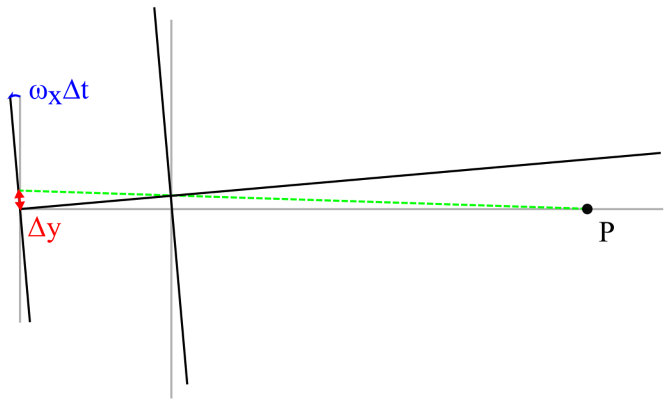

Figure 3.3: The effect of rotating the pinhole camera around the x-axis

Assume the camera has been rotated around its x-axis between two consecutive frames (figure 3.3). The displacement in the image plane induced by the rotation is

$$\Delta y = f\tan(\omega_x \Delta t), \tag{3.10}$$

where ω_x is the angular velocity (which can be obtained from a gyroscope) and Δt is the time between two consecutive frames. Similarly, a rotation around the y-axis induces a displacement in the x-axis:

$$\Delta x = f\tan(\omega_y \Delta t). \tag{3.11}$$

Displacements Δx and Δy have to be subtracted from the resulting optical flow in order to compensate the angular motion. Rotation around the optical axis (z-axis) does not have to be corrected as the induced optical flow is useful for the estimation of the vehicle's heading. Equation 3.9 becomes then

$$\Delta X = -[\frac{s}{f}u - f\tan(\omega_y \Delta t)] \cdot Z, \tag{3.12}$$

$$\Delta Y = -[\frac{s}{f}v - f\tan(\omega_x \Delta t)] \cdot Z. \tag{3.13}$$

3.3 H.264 video format

The H.264 Advanced Video Coding standard [Ric10] defines a syntax for encoded video and a method for its decoding.

Frames are divided into one or more slices which contain sets of macroblocks. Each macroblock represents a 16×16 pixel partition of a frame. The standard uses a prediction mechanism to reduce redundancy in the encoded information by utilizing similarities between different image parts (within the same frame or between different frames). Three types of macroblocks are defined:

- I-type macroblock uses a prediction from neighboring samples in the same frame (intra prediction)
- P-type macroblock uses a prediction from samples in a previously encoded frame (inter prediction). This might be the "past" or the "future" frame depending on encoding order
- B-type macroblock use a prediction from up to two previously encoded frames (inter prediction)

Each macroblock is formed by a residual image and parameters of the used prediction method. For the inter prediction mode, the macroblock contains a residual image (the difference between the reference macroblock and the actually encoded image), source information (i.e. to which frame the reference macroblock belongs) and a motion vector describing the displacement of the reference macroblock with respect to the actually encoded image. Macroblocks can be further divided into smaller rectangular partitions with different prediction sources and therefore with different motion vectors.

Motion vectors encoded in P-type macroblocks could be used as an estimation of the optical flow required for the ego-motion estimation. This, however, depends on the actual configuration of the encoder as the standard specifies multiple profiles which include different methods. The encoder might process frames in an arbitrary order. Furthermore, the ordering of encoded frames might differ from the frame display ordering.

Chapter 4
Design

This chapter describes the proposed method of the ego-motion estimation using a ground-facing camera. An efficient mixed CPU and GPU solution related to the actual hardware and a fully software solutions are proposed. As depicted on figure 4.1, the solution can be broken into separate steps.

First, an optical flow is obtained either by utilizing the motion vectors provided by the hardware CME block or by using alternative software methods. The optical flow is then corrected for the effects induced by roll and pitch rotations of the camera. The ego-motion is then estimated using a robust RANSAC scheme and scaled to real-world units.

As the angular correction and scaling have been already described in section 3.2, this section focuses on methods for obtaining the optical flow and estimating motion.

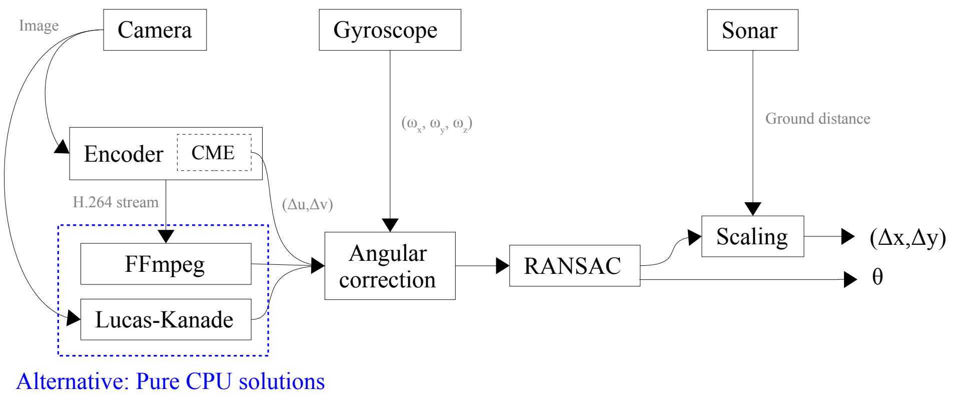

Figure 4.1: A pipeline of the designed solution

4.1 Mixed CPU and GPU solution

The Raspberry Pi computer includes a VideoCore IV multimedia processor which is able to encode and decode video efficiently as it contains an array of GPU units and an instruction set suitable for digital signal processing. The mixed solution is based on the possibility to obtain motion vectors estimated by a hardware H.264 encoder implemented in the VideoCore and use them as the optical flow required to estimate the ego-motion.

4.1.1 Hardware H.264 encoder

The encoder implemented in the VideoCore DSP uses a “Low-delay prediction structure” (figure 4.2) which is suitable for an embedded system with constrained resources as it minimizes memory requirements and delay. The first encoded frame is always encoded as an I slice as there are no previously encoded frames suitable for inter prediction. Subsequent frames are encoded as P-slices with a prediction source fixed to the “past” frame [bfn14a]. The encoder does not support B-slices [bfn14b]. The encoder inserts additional I-frames to the stream to minimize transmission errors. This also enables the peer device to decode stream which has already started in the past.

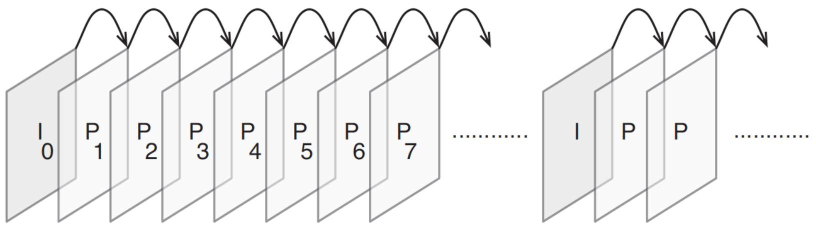

Figure 4.2: Ordering of encoded frames when using the Low-delay prediction (from [Ric10])

I have performed an experiment which involved the decoding of a 30-second 480×480 px video sequence recorded by the RaspiVid application at 30 FPS with the default encoding options to verify the assumptions. As expected, the sequence did only include P-frames with an I-frame inserted after every 60 P-frames. The distribution of I-frames in time is depicted on figure 4.3 (note that the recording application did not record a full 30-second sequence as its timeout functionality is not related to the number of frames actually captured). While the first frame will always be an I-frame, the insertion of additional I-frames can be suppressed by setting the `-intra` option of the RaspiVid application to zero. This is important as the I macroblocks do not use intra-frame prediction and are therefore useless for the intended application.

The experiment has also shown that all motion vectors represent motion prediction with respect to the “past” frame.

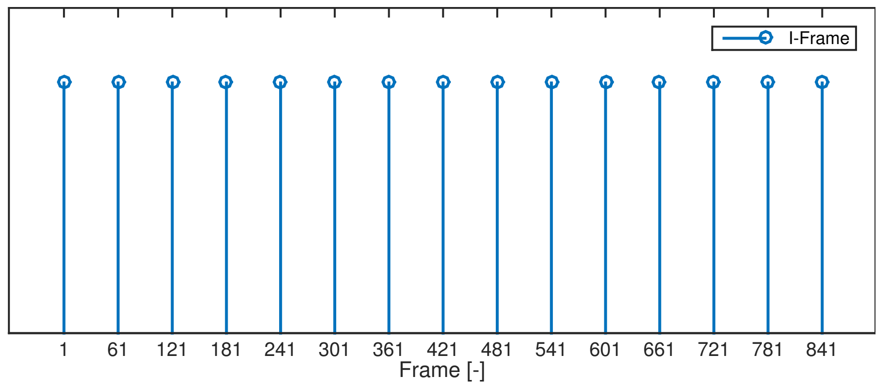

Figure 4.3: Distribution of I-type frames in the bitstream

Motion vectors from coarse motion estimation block

The VideoCore uses two hardware motion estimation blocks for video encoding. A coarse motion estimation (CME) block estimates displacement in pixel resolution and a subsequent fine motion estimation (FME) block is able to estimate the displacement in a sub-pixel resolution [bfn14a].

The motion estimation block uses a block matching method (described in section 3.1.2) to estimate the displacement $(\Delta u, \Delta v)$. For each macroblock in the current frame, the closest match is found in the previous frame within a given range (figure 4.4). Vectors from the CME block can be obtained directly from the encoder while vectors from the FME block are encoded in the final H.264 bitstream [bfn15].

For each P-frame, the encoder provides a buffer which contains a single 32-bit value for each 16×16 px macroblock [Upt14, Hol14]. The most significant 16 bits represent a Sum of Absolute Differences (SAD) value. The SAD value is a measure of the estimated motion's quality: the lower the SAD, the better match has been found. The other 16 bits represent motion in horizontal and vertical directions (8-bit signed integer per direction).

The number of macroblocks provided by the CME is constant for each frame. This is different from macroblocks encoded in the H.264 bitstream which are often divided to sub-macroblocks (as described in section 3.3). The buffer provided by the encoder for each I-frame is internally set to zero [bfn14a] as I-type macroblocks do only provide the inter-frame prediction.

Although the displacement can theoretically be in range ± 127 pixels, a closer analysis shows it is in fact in range ± 64 pixels from macroblock's center. Figure 4.5 shows the histogram of vector displacement in an experimental video sequence. The video sequence involved rapid movements of the scene to yield large macroblock displacements.

Moreover, the analysis shows that the CME, in fact, estimates motion in two-pixel resolution (i.e. only even values are present).

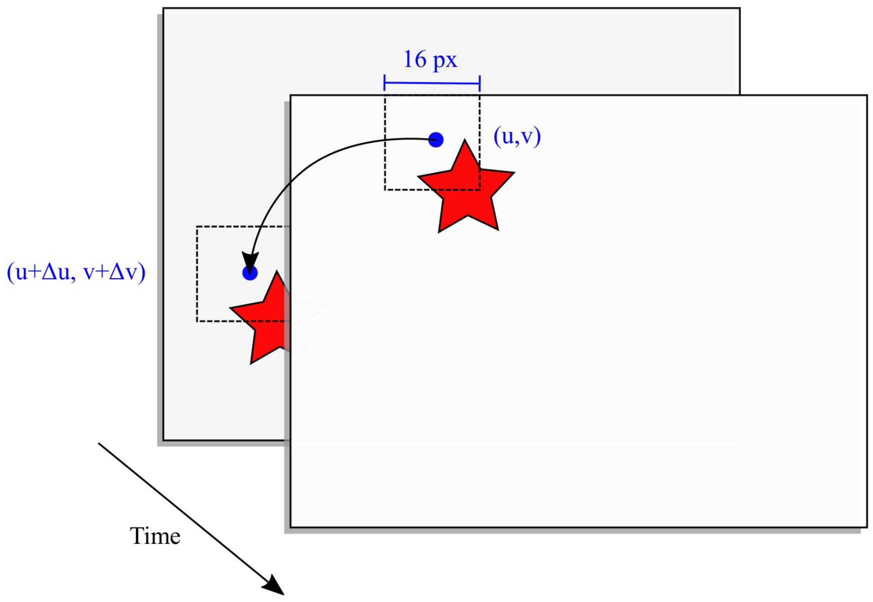

Figure 4.4: Illustration of the motion vector estimated for a selected macroblock

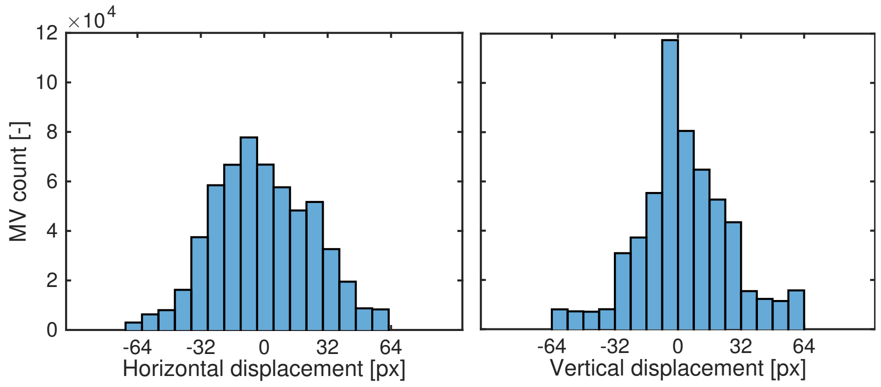

Figure 4.5: Displacement of non-zero motion vectors from the CME block

4.1.2 Motion estimation using the RANSAC algorithm

RANSAC (Random Sample Consensus) [FB81] is a popular algorithm for estimating model parameters from data with a large portion of outliers. Unlike traditional approaches such as the least squares method which use a large number of data samples to estimate parameters (leading to errors caused by a likely presence of outliers in the selected sample), the RANSAC uses the minimum number of data samples required for the estimation.

In each iteration, the algorithm 1 estimates model parameters using a small number of data samples. The estimated model is then applied to all data samples in order to identify inliers (i.e. samples which fit the model with a predefined tolerance). If the proportion of inliers exceeds given threshold, the model is re-estimated using all identified inliers and the algorithm terminates. Otherwise, the algorithm iterates up to N times and finally estimates parameters for the largest set of inliers found.

Algorithm 1 RANSAC

```
Inputs:  Maximal number of iterations $N$, set of data samples $S$, number
         of samples required to estimate model parameters $s$, desired
         proportion of inliers $\Omega$
Output:  The best set of parameters $\boldsymbol{\theta}^*$

- $k := 0, S_i^* := \emptyset$
- Repeat until $k = N$:
    - $k := k + 1$
    - Select random sample $S_k \subset S, |S_k| = s$
    - Estimate parameters $\boldsymbol{\theta} := f(S_k)$
    - Determine the set of inliers $S_i$, $\omega := \frac{|S_i|}{|S|}$
    - If $\omega \geq \Omega$: Re-estimate parameters using all inliers and terminate,
      i.e. $\boldsymbol{\theta}^* := f(S_i)$
    - If $|S_i| > |S_i^*|$: $S_i^* := S_i$
- Estimate parameters using the largest set of inliers and terminate,
  i.e. $\boldsymbol{\theta}^* := f(S_i^*)$
```

Model for the RANSAC algorithm

The estimated optical flow $(\Delta u, \Delta v)$ represents a displacement and rotation between two subsequent image frames. This transformation can be seen as an affine transformation with rotation, uniform scaling, and translation components

$$\boldsymbol{R} = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}, \boldsymbol{S} = \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}, \boldsymbol{t} = \begin{bmatrix} t_u \\ t_v \end{bmatrix}. \tag{4.1}$$

The relation between macroblock's center (u, v) in the current image and a matching macroblock's center $(u + \Delta u, v + \Delta v)$ in the previous image (figure 4.4) is then

$$\begin{bmatrix} u \\ v \end{bmatrix} = \boldsymbol{RS} \begin{bmatrix} u + \Delta u \\ v + \Delta v \end{bmatrix} + \boldsymbol{t} = \begin{bmatrix} s\cos\phi & -s\sin\phi \\ s\sin\phi & s\cos\phi \end{bmatrix} \begin{bmatrix} u + \Delta u \\ v + \Delta v \end{bmatrix} + \begin{bmatrix} t_u \\ t_v \end{bmatrix}. \tag{4.2}$$

The model is therefore represented by four parameters

$$\boldsymbol{\theta} = \begin{bmatrix} s\sin\phi & s\cos\phi & t_u & t_v \end{bmatrix}^T, \tag{4.3}$$

which can be estimated from two samples $(\Delta u_1, \Delta v_1)$ and $(\Delta u_2, \Delta v_2)$ using the least squares method:

$$\begin{bmatrix} u_1 \\ v_1 \\ u_2 \\ v_2 \end{bmatrix} = \begin{bmatrix} -v_1 - \Delta v_1 & u_1 + \Delta u_2 & 0 & 1 \\ u_1 + \Delta u_1 & v_1 + \Delta v_1 & 0 & 1 \\ -v_2 - \Delta v_2 & u_2 + \Delta u_2 & 1 & 0 \\ u_2 + \Delta u_2 & v_2 + \Delta v_2 & 0 & 1 \end{bmatrix} \boldsymbol{\theta} \tag{4.4}$$

Note the rotation ϕ does not depend on the scaling factor s:

$$\phi = \tan^{-1}\frac{s\sin\phi}{s\cos\phi} = \tan^{-1}\frac{\theta_1}{\theta_2}. \tag{4.5}$$

Required number of iterations

As described in [HZ03], for a given proportion of inliers ω the required number of iterations N can be calculated in advance so that the probability P that a sample of size s is an uncontaminated set (i.e. free of outliers) is reasonably high (equation 4.6).

$$N = \frac{\log(1-P)}{\log(1-\omega^s)} \tag{4.6}$$

As seen from figure 4.6, the number of samples required to estimate the proposed model is relatively low even if the assumed proportion of inliers is just 10%.

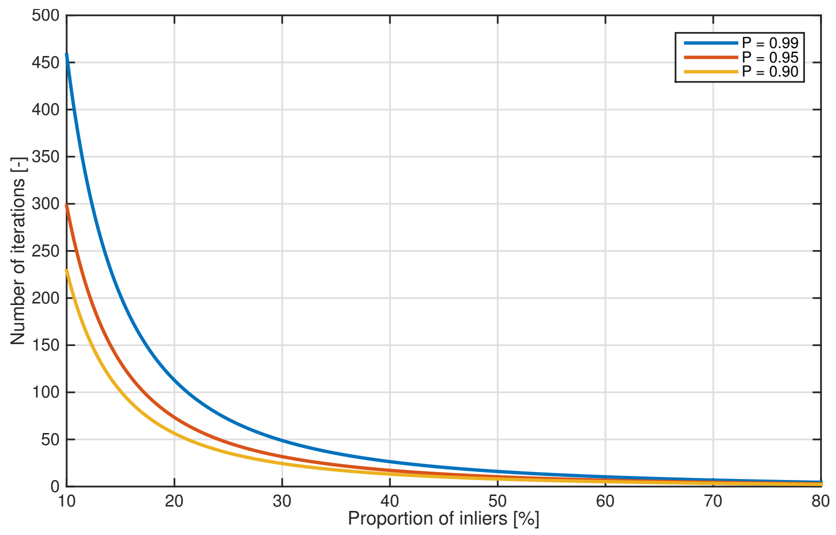

Figure 4.6: Number of RANSAC iterations required for model estimation

4.2 Pure CPU solution

The pure CPU solution is an alternative to the the Mixed GPU and CPU solution described in section 4.1. As the mixed solution is highly dependent on the actual hardware and firmware of the Raspberry Pi computer, it is beneficial to propose an alternative solution for similar embedded platforms where the encoder is not present or where the firmware does not provide any output from the motion estimation block.

This section focuses on alternative methods to obtain the optical flow, either using optical flow algorithms or by a partial bitstream decoding. While there are other methods to estimate motion between two consecutive image frames such as phase correlation, these are usually too complex for a typical embedded application.

4.2.1 Block matching algorithm

The block-matching algorithm described in section 3.1.2 is a natural candidate for the pure software implementation because it closely resembles the process being performed by the CME block and because it can be parallelized. Moreover, SIMD instructions provided by the ARM NEON extension would enable efficient processing. The resulting flow could be also refined to a sub-pixel resolution in a way similar to the PX4Flow sensor.

I have however decided to let this option unimplemented because the main aim of this work is the development of the Mixed GPU and CPU solution and because the software implementation of the block-matching algorithm does not represent any improvement over the existing sensor.

4.2.2 Lucas-Kanade algorithm

The Lucas-Kanade algorithm seems to be ideal for an embedded system as it only computes the optical flow for a set of pixels instead of the full image. Also, the process can be well parallelized.

However, as mentioned in section 3.1.2, the algorithm only works for small displacements in the image. This problem is usually addressed by the usage of so-called image pyramids. Each level of the pyramid is formed by a down-sampled version of the original image. The algorithm then begins its search on the lowest resolution level and refines its search on higher-resolution levels.

This approach is described in [Bou01]. The author also proposes a method for tracking features between samples and a method for declaring a feature lost, so that it is not necessary to run the feature detector for each frame. Furthermore, the pyramids can be reused between consecutive frames which decreases the overall computation time.

The algorithm can also be used to enhance the mixed GPU and CPU solution. As discussed in section 4.1.1, the motion vectors only provide two-pixel resolution. Instead of detecting image features, the motion vectors could be used as an initial guess for the Lucas-Kanade tracker, which would refine them to sub-pixel resolution. This would also eliminate the need for pyramids as only small differences are expected between original and refined motion vectors.

4.2.3 Partial bitstream decoding

The optical flow can also be obtained by decoding the encoded bitstream. As this solution does not rely on any hardware-specific features (such as the access to the CME block), it can be considered a CPU solution. In order to obtain the motion vectors associated with each macroblock, it is necessary to use a software decoder. However, the decoding usually presents a significant load to the CPU.

As described in [YSK09], the computational time can be significantly reduced by decoding the H.264 bitstream only partially. The authors propose a method for identifying macroblocks to be encoded in order to detect moving objects on a static background. This approach is usable for systems processing data from a large number of surveillance cameras in a real-time.

A similar approach might be also used for the intended application. In this case, all P-type macroblocks are useful (as they include the motion vectors) but the image information is not required. This allows the decoder to skip some operations such as the image reconstruction using the inverse DCT.

Chapter 5

Experimental setup

This chapter describes the hardware and software components used for and implementation of the proposed solution.

5.1 Hardware

The hardware selection is heavily affected by the assignment. Raspberry Pi is a single-board computer which has gained massive popularity in recent years. Its main advantages are good availability, low cost and an embedded video acquisition and compression pipeline. It comes in multiple form factors ranging from the smallest model Zero to the largest model 3B.

I have selected the most recent Raspberry Pi 3 which features a 1.2 GHz quad-core ARM Cortex-A53 CPU with 1 GB RAM. The CPU supports an SIMD instruction set called NEON which is useful for intended computer vision tasks.

While the Zero or Compute Module boards are smaller and therefore more suitable for the intended application, they are based on an older generation of Broadcom's SoC with pre-Cortex ARM11 CPU which does not support the NEON instruction set.

5.1.1 Camera

The Raspberry Pi computer is equipped with a CSI port for interfacing camera chips. The image acquisition process involves steps (camera configuration, lens shading correction, bayer filter, image distortion, etc.) which are processed by the the VideoCore DSP to reduce CPU load.

Because the relevant firmware is a closed source binary blob, the camera selection is effectively limited to two image sensors. This also means that the image acquisition pipeline can not be modified in any way (e.g. altering the lens shading pattern for a different lens or using a custom camera calibration).

Currently, there are two available camera modules produced for the Raspberry Pi, an OmniVision OV5647-based “v1” board and a Sony IMX219-based “v2” board introduced in 2016. I have selected the latter as the OV5647 sensor used by the “v1” module has already reached its end-of-life date at the end of 2014.

The IMX219 sensor is a back-illuminated rolling shutter CMOS sensor aimed at consumer electronics. Parameters of the selected camera module are listed in table 5.1.

Resolution:	3280×2464 pixels
Focal length:	3.04 mm
Sensor size:	3.674×2.760 mm
Pixel size:	$1.2 \times 1.2\,\mu$m
FOV:	$62.2° \times 48.8°$

Table 5.1: Parameters of the Raspberry Pi Camera Module v2 [eLi16]

The image captured by the sensor has to be resampled to the desired resolution. According to [Jon16], the image acquisition pipeline uses a few discrete video modes. The Modes which employ binning are listed in table 5.2. The binning is performed directly by the image sensor and reduces noise in the final image, which is useful especially in low light conditions. If the desired resolution differs from a resolution provided by the selected video mode, an additional scaling is performed by the VideoCore DSP. I have selected mode 4 because is uses the full field of view and works at lower frame rates.

Mode	Cropped area	Resolution	Frame rate
4	3280×2464 pixels (full)	1640×1232 pixels	0.1 to 40 FPS
5	3280×1844 pixels	1640×922 pixels	0.1 to 40 FPS
6	2560×1440 pixels	1280×720 pixels	40 to 90 FPS
7	1280×960 pixels	640×480 pixels	40 to 90 FPS

Table 5.2: Available video modes with the highest frame rate

The comparison shows that the machine vision sensor used by the PX4Flow is better as it features larger pixels and up to 4×4 binning. While it is possible to choose a USB-connected camera with different parameters, the necessity to involve CPU in the image acquisition process would reduce possible frame rate and increase CPU load. It would also make the overall mechanical design bulkier which is not practical with respect to the intended application.

Calibration and theoretical limits

The theoretical maximum velocity can be obtained using modified equation 3.9. The maximum velocity $\dot{x}_{max}$ $[\mathrm{m \cdot s^{-1}}]$ depends on the height above ground (the higher the altitude, the higher the speed) and the frame rate n_{FPS}:

$$\dot{x}_{max} = \frac{b \cdot s}{f} \cdot u_{max} \cdot n_{FPS} \cdot Z \tag{5.1}$$

The term u_{max} is the maximum displacement detectable by the block matching algorithm which is 64 pixels according to section 4.1.1. The pixel size s must be multiplied by the scaling factor b. However, the scaling factor is unknown as the only given information is that the original image is binned to half by the sensor before being resized by the DSP to the desired resolution. The resizing process is unknown and might possibly include any combination of scaling, line skipping, and cropping. The factor b must be therefore estimated using a camera calibration process.

The camera parameter matrix $\boldsymbol{K}$ obtained for the 480×480 px resolution is

$$\boldsymbol{K}\,[\mathrm{px}] = \begin{bmatrix} \frac{f}{p_w} & 0 & u_0 \\ 0 & \frac{f}{p_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \approx \begin{bmatrix} 531.97 & 0 & 239.50 \\ 0 & 531.90 & 239.50 \\ 0 & 0 & 1 \end{bmatrix},$$

where p_w and p_h are the resulting image's pixel width and height and (u_0, v_0) is the projection centre. Because the image sensor has square pixels of size s, the factor b can be estimated from averaged p_w and p_h and known focal length f as

$$b = \frac{p_w + p_h}{2s} = \frac{\frac{f}{k_{1,1}} + \frac{f}{k_{2,2}}}{2s} \approx 4.7625.$$

The theoretical minimum velocity can be calculated using equation 5.1 for $u_{max} = 2$. The theoretical minimum and maximum velocity detectable by the sensor is depicted in figure 5.1 and table 5.3. The sensor will only provide valid measurements for velocities within the given range. The figure also contains the theoretical maximum velocity detectable by the PX4Flow sensor for comparison (its theoretical minimum detectable velocity is unknown).

The factor b is different for frame rates above 40 FPS as a different video mode must be used. For a video mode 7 the factor can be easily estimated as $b \approx \frac{2 \cdot 960}{480} = 4$.

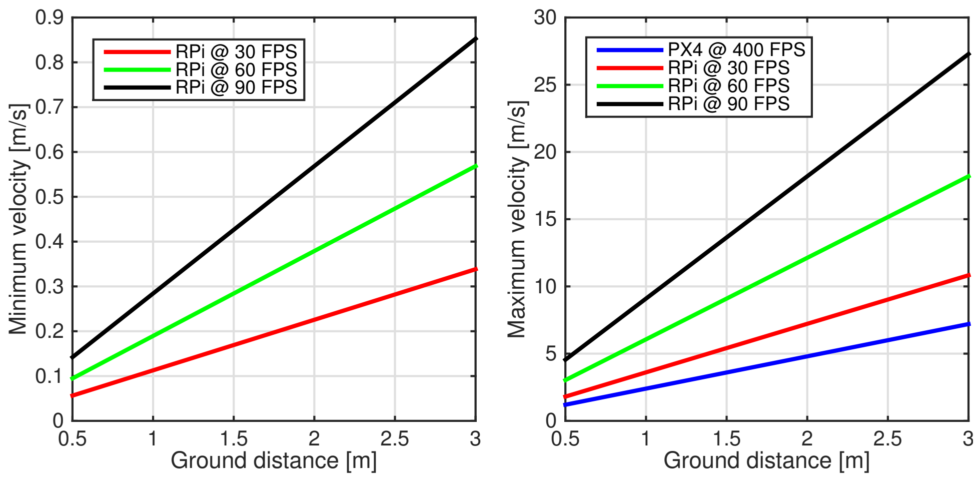

Figure 5.1: Theoretical velocity limits with respect to the ground distance

Ground distance [m]	0.5	1	3
Minimum velocity $[\mathrm{cm} \cdot \mathrm{s}^{-1}]$	5.6	17.9	33.8
Maximum velocity $[\mathrm{m} \cdot \mathrm{s}^{-1}]$	1.8	3.6	10.8

Table 5.3: Theoretical velocity limits for the frame rate of 30 FPS

5.1.2 Distance sensor

The distance sensor is required to measure the distance to the ground so that it is possible to convert the optical flow in pixels into real-world units (eq. 3.9). I have selected the MaxBotix HRLV-EZ4 [Max14] ultrasonic sensor which has a very narrow beam when compared to other available ultrasonic sensors. The PX4Flow module uses the same sensor.

The sensor measures the distance periodically and returns the measured value in millimeters over a serial interface. As it is possible to power the sensor by 3.3V supplied by the Raspberry Pi, it is not necessary to convert UART voltage levels.

The maximum range detectable by the sensor is 5 meters, which is sufficient for the intended usage. For applications where a higher range is required a laser sensor such as the Garmin LIDAR-Lite v3[1] might be an option. It would be also possible to use a barometric pressure sensor which would require calibration at the ground level.

5.1.3 Gyroscope

The gyroscope is required to measure the angular velocity between consecutive frames so that it is possible to subtract angular corrections from the resulting optical flow (eq. 3.12).

I have selected the L3GD20H [STM13] MEMS 3-axis digital gyroscope by STMicroelectronics, which is the successor of the older L3GD20 chip used by the PX4Flow sensor. The sensor is connected to the Raspberry Pi using an I^2Cinterface. The chip provides a number of features, such as an integrated temperature sensor, a FIFO buffer for the sampled data samples, selectable gain or low-pass and high-pass digital filters.

[1]https://buy.garmin.com/en-US/US/p/557294

5.2 Software

The software selection (especially the operating system) is also strongly coupled to the Raspberry Pi computer and the features of the ARM Cortex-A processor.

5.2.1 Raspbian

Being a popular project, the Raspberry Pi is supported by a wide range of operating systems, including Linux-based distributions, BSD, Android, Windows or Plan9. I have selected a Raspbian Linux-based distribution which is officially supported by the Raspberry Pi Foundation.

The Raspbian [TG12] is an unofficial port of the Debian Wheezy distribution for a hard-float ARM architecture. Although it had originated as an unofficial and independent project, the Foundation adopted it and periodically releases[2] own system images (which include custom code such as the interface for the VideoCore DSP). New releases include a “Lite” variant which does not contain any unnecessary packages such as an integrated development environment or a desktop environment.

5.2.2 OpenCV

OpenCV [Bra00, Its15] is a free multi-platform library for common tasks related to computer vision. It is written in C and C++ and provides bindings to other languages such as Python. I have selected it because it provides optimizations for the ARM Cortex-A architecture, which include the usage of intrinsics for NEON SIMD instructions and usage of the TBB parallelization framework by Intel.

The OpenCV provides both low-level functionality (such as linear algebra) useful for the proposed mixed CPU and GPU solution described in section 4.1 and more complex algorithm implementations (such as feature detection or Lucas-Kanade) useful for the pure CPU solution described in section 4.2.

The codebase receives lots of contributions with varying quality, which results in difficulties for maintainers and relatively low periodicity of releases. I have used version 3.1 from December 2015, which was the latest stable release available at the time of development.

5.2.3 Image acquisition

As described in chapter 4, the proposed solution requires both encoded video data (which contain inline motion vectors) and raw image data for image processing.

The Raspbian distribution provides a standard Video4Linux driver which enables the camera to be used with various third-party applications, such as

[2] https://www.raspberrypi.org/downloads/raspbian/

the OpenCV framework. However, the driver does not provide any motion vectors.

Another way is to use the vendor's application interface to access the camera and encoder. Broadcom uses a Multimedia Abstraction Layer (MMAL) to enable interaction between the CPU and so-called components in the VideoCore DSP. However, this interface lacks proper documentation (only header files and example code is available), which makes the development complicated.

Libraries

The most advanced available MMAL wrapper is the Picamera module for Python [Jon13]. It is actively maintained and well documented, works directly with the MMAL interface and has the richest set of options. However, the usage of Python would be a bottleneck in an embedded application.

Another available solution is the Raspicam C++ library [Sal13] which features a camera class for the OpenCV. It uses the MMAL API to provide a raw image from the camera and allows a limited set of configurations. The code is, however, not actively maintained and lacks newer features. It also lacks the ability to provide the encoded image and motion vectors.

RaspiVid

The Raspbian distribution contains a set of programs for image acquisition which are open-sourced. The RaspiVid [Hug13] program allows the user to record encoded video and provides a rich set of camera parameters. It also optionally provides inline motion vectors from the coarse motion estimation block. The encoded video is insufficient for intended computer vision tasks as its decoding would introduce a significant CPU load.

The RaspiVidYUV is a copy of the RaspiVid which produces raw video in YUV format. It lacks the ability to store motion vectors as the encoder stage is omitted. Both programs are being actively developed both by the community and employees.

As there was no available solution to fulfill the requirements at the time of writing, I have decided to modify the RaspiVid program to provide both raw video and motion vectors (described in section 6.1).

Chapter 6

Implementation

This chapter describes implementation details of the proposed solution described in chapter 4. The software is written in C and C++.

6.1 RaspiVid modification

As discussed in section 5.2.3, I have decided to modify the existing RaspiVid application to obtain both image data and motion vectors from the CME block. The application is based on Broadcom's proprietary MMAL API. The API is used to set up components and their interconnections and to access them from the CPU. These components run completely in the VideoCore DSP and the CPU only handles callbacks which provide access to the input our output buffers. The original configuration of the components is depicted in figure 6.1.

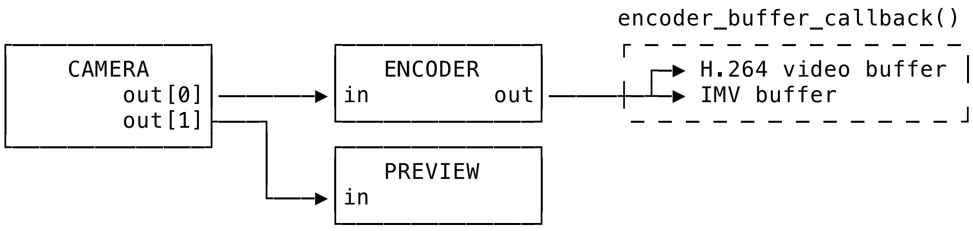

Figure 6.1: Diagram of the original RaspiVid MMAL architecture

This configuration allows processing of inline motion vectors but does not allow image processing. As both camera's video outputs are routed to components, it is not possible to assign a callback to allow the CPU to access image data. I have therefore added a VIDEO_SPLITTER component which has the ability to split video streams to multiple outputs. The VIDEO_SPLITTER performs format conversion to grayscale so it is not necessary to configure the format at the camera's output (the camera's output format is optimized for the most efficient encoding). The new configuration is depicted in figure 6.2.

I have added two options to the RaspiVid application. Option `-raw` allows the user to save the raw video into a file. The raw video output format (RGB, YUV or grayscale) can be configured using a `-raw-format` option. These mod-

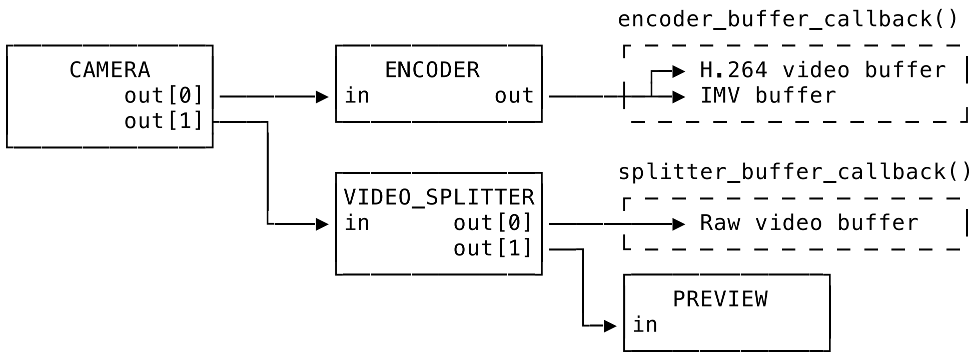

Figure 6.2: Diagram of the modified RaspiVid MMAL architecture

ifications have been contributed into the project's repository and integrated[1]. This extra effort simplifies patching of future RaspiVid updates so that they can be used in the proposed application: both `encoder_buffer_callback()` and `splitter_buffer_callback()` contain a single line code which passes buffers to the main application for further processing.

6.2 Camera calibration

I have used an automated camera calibration code which is a part of the OpenCV library [Gá11] to calibrate the camera. The application allows the user to calibrate the camera either by providing a sequence of images of the calibration pattern (e.g. checkerboard) or by providing a live stream from the camera. While it might be possible to use the existing Video4Linux driver to obtain the stream, it is not clear what video mode (table 5.2) it uses.

The calibration makes only sense for the same configuration of scaling and cropping which is used by the final application. I have therefore modified the code to grab images from the Raspberry Pi camera in the way described in section 6.1. As the code might also be useful for other users, I have released it as a separate project[2].

The radial distortion estimated by the calibration process is also used to undistort motion vectors before being compensated for the effects of camera rotation.

6.3 RANSAC

The OpenCV library provides a function `estimateRigidTransform()` which is a part of its "video" module. The function accepts two sets of 2D point correspondences or two images and finds an affine transformation between them (either a full affine transform without any restriction or the $[\boldsymbol{RS}|\boldsymbol{t}]$ transform restricted to uniform scale, rotation, and translation as described

[1]https://github.com/raspberrypi/userland/pull/342
[2]https://github.com/adamheinrich/RaspiCalib

in section 4.1.2). In case two images are provided as the input, the function first finds the corresponding points and computes optical flow using the Lucas-Kanade algorithm.

The affine transform is estimated using the RANSAC scheme described in algorithm 1. Parameters of the algorithm (e.g. the maximum number of iterations $N = 500$ or the required proportion of inliers $\Omega = 0.5$) are however hard-coded[3] and can't be changed. Moreover, the function seems to always draw three samples required by the estimation of the full affine transform even if the model is restricted to $[\boldsymbol{RS}|\boldsymbol{t}]$.

This clearly increases the theoretical maximum run time: The number of iterations (equation 4.6) required to draw at least one uncontaminated sample with the probability $P = 0.99$ is 35 for sample size $s = 3$ but just 17 for the sample size $s = 2$.

I have therefore implemented a custom function which uses the transform matrix estimation from the original `estimateRigidTransform()`. The function is parallelized using a OpenCV wrapper for the TBB library, and the number of iterations is set to a fixed value required to draw at least one uncontaminated sample for a data set with 15% of inliers.

6.4 Pure CPU solution

I have used functions[4] provided by the OpenCV library to evaluate the Lucas-Kanade method proposed in section 4.2.2.

I have used the FFmpeg [dev16] library to implement the partial bitstream decoding method proposed in section 4.2.3. The library provides access to motion vectors recovered during the decoding process (`+export_mvs` option). The decoding process is made faster by configuring the decoder to skip some image reconstruction steps such as the IDCT transform.

The decoded motion vectors represent the displacement of motion blocks between two consecutive frames as described in section 4.1.1, they are however represented as integers and therefore do not provide sub-pixel accuracy (which would be an improvement over vectors recovered from the CME block). Also, the SAD value or a similar quality measure is not present.

Because the decoding is being processed by the CPU, this method is slower than the mixed solution. It is, however, useful for platforms which include hardware encoder but do not have access to the CME block.

6.5 Integration

To be able to send measured data to a host controller, a communication protocol must be implemented.

The PX4Flow sensor uses a MavLink protocol. MavLink is an open protocol which provides a set of message types designed specifically for

[3]https://github.com/opencv/opencv/blob/3.1.0/modules/video/src/lkpyramid.cpp#L1354

[4]`goodFeaturesToTrack()`, `buildOpticalFlowPyramid()` and `calcOpticalFlowPyrLK()`

the communication with micro aerial vehicles as well as for the internal communication between controllers, sensors and other peripherals used by MAVs. It is used in a wide range of systems and application, such as the ArduPilot[5] or QGroundControl[6]. An open-source C implementation [Mei09] for the serialization and deserialization of messages is available.

I have selected the MavLink protocol to maintain the highest possible compatibility with the PX4Flow sensor. I have however decided to use a UDP socket instead of UART as a transport layer. As the Raspberry Pi's only UART interface is used for the ultrasonic sensor, it is not possible to use the serial interface for MavLink communication without any additional hardware (such as the external USB converter or a I^2Cor SPI to UART bridge chip). Moreover, the UDP can also be used to communicate with other programs running on the Raspberry Pi computer.

The system therefore periodically sends the same message types as the PX4Flow sensor:

- OPTICAL_FLOW_RAD (translational velocity, change in orientation, angular velocity from the gyroscope, and ground distance)
- OPTICAL_FLOW (translational velocity in pixels and meters per second, ground distance) – this is an "older" message type maintained for backward compatibility with previous PX4Flow versions
- HEARTBEAT

I have tested the implemented MavLink integration with the QGroundControl application. The measured data can also be logged into a semicolon-separated CSV file.

6.6 Data loggers

I have implemented two data loggers to be able to log data from reference sensors during evaluation.

The GPS logger receives standard NMEA sentences from a GPS receiver connected to the Raspberry Pi via a USB to UART converter. The received data are logged into a text file for later processing. The text file is then converted to a comma-separated list of geographic coordinates using the online tool GPS Visualizer [Sch02].

The PX4Flow logger receives MavLink messages sent over the USB interface. The received messages are parsed and logged into a semicolon-separated CSV file together with a timestamp. As the USB option is in fact UART over USB, this program can be easily modified to log data from a sensor connected over UART as well. As the logger might also be useful for different use cases, I have released it under an open-source license as a separate project[7].

[5]http://ardupilot.org/
[6]http://qgroundcontrol.org/
[7]https://github.com/adamheinrich/mavlog

6.7 Hardware and mechanical construction

The mechanical construction went through multiple iterations. Early prototypes were based around a clear acrylic case with mounting holes designed for the camera module. Sensors were connected using a wire-wrapping technique and glued directly to the acrylic case. The construction is depicted in figure 6.3 (note that it uses a larger ultrasonic distance sensor).

The early prototype was however not suitable for outdoor experiments. I have therefore designed a more firm construction based around a prototyping PCB for the Raspberry Pi. The PCB features mounting holes for the camera and sensors which increase mechanical robustness of the construction. The placement of components and lengths of distance screws have been designed to fit the original Raspberry Pi plastic case. The final construction is depicted in figure 6.4.

All required components are listed in table 6.1.

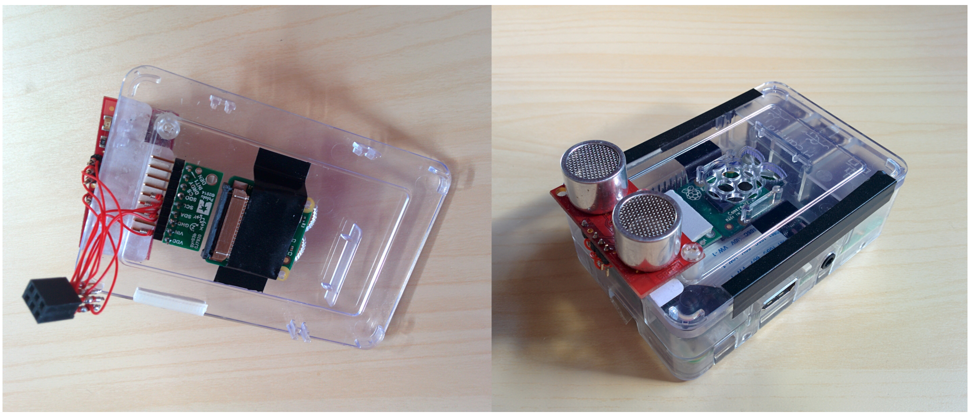
Figure 6.3: Photo of the early prototype

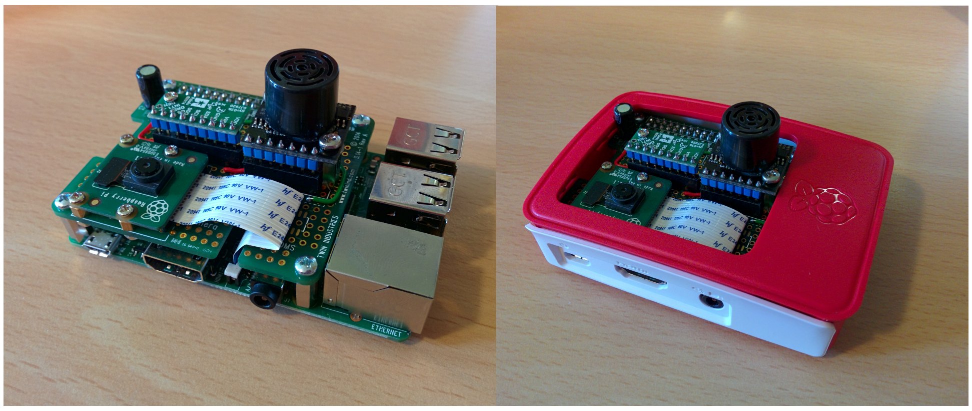
Figure 6.4: Photo of the final construction

Component name	Component usage
Raspberry Pi 3 Model B[8]	Single-board computer
Raspberry Pi Camera Module v2[9]	Camera
Raspberry Pi Case[10]	Plastic box
8GB micro SD	Memory card
MaxBotix HRLV-EZ4 (MB1043)[11]	Ultrasonic sensor
Pololu L3GD20H Carrier (#2129)[12]	Gyroscope with a breakout PCB
Twin Industries 3.14-1[13]	Prorotyping PCB

Table 6.1: Bill of materials

[8] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[9] https://www.raspberrypi.org/products/camera-module-v2/
[10] https://www.raspberrypi.org/products/raspberry-pi-case/
[11] http://www.maxbotix.com/Ultrasonic_Sensors/MB1043.htm
[12] https://www.pololu.com/product/2129
[13] http://twinind.com/index.php/products/314/314-1/

Chapter 7

Evaluation

This chapter describes evaluation of the proposed solution in both indoor and outdoor environment. Because the tests have been performed before the finalization of the software, some steps (such as the translational velocity scaling into real-world units) have been performed in post-processing.

7.1 Indoor testing

Indoor tests have been performed in a small arena equipped with a camera-based localization system WhyCon [Piv16] which was used as a ground truth for comparisons. The system is able to record the position of the measured object (detected using special circular markers) with up to six degrees of freedom. I have however decided to use a 2D mode which is faster and therefore provides more data samples.

The evaluated sensor was placed on a manually moved wheeled table (depicted on figure 7.1) to maintain a constant altitude of 46 cm. The table was moved manually.

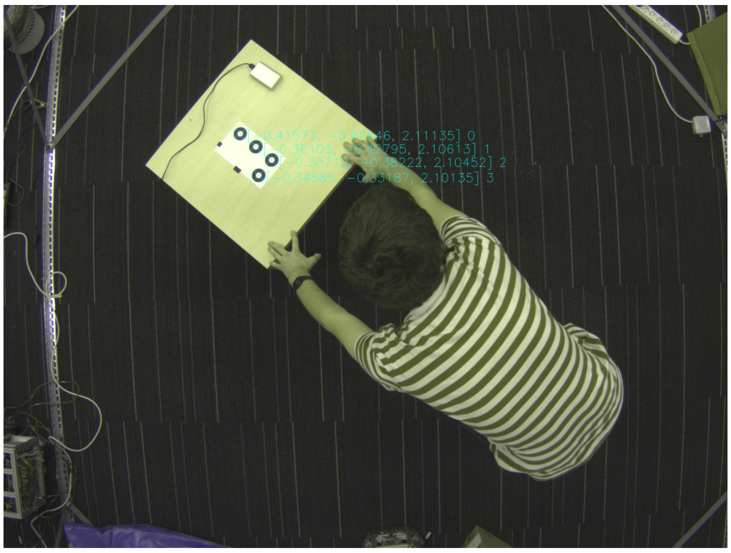

Figure 7.1: The table with attached circular markers as seen by the WhyCon

7.1.1 Mixed CPU and GPU solution

I have used two trajectories to evaluate the velocity measurement. The "Squares" trajectory is depicted on figure 7.2. The second trajectory, "Circles", is similarly formed by overlapping circles. Figure 7.3 shows velocities measured by the Raspberry Pi compared to derived and filtered positions measured by WhyCon. Average errors and standard deviations are shown in table 7.1.

To evaluate orientation estimation, I have used the WhyCon system in 6DOF mode. Figure 7.4 shows the computed trajectory and the resulting orientation compared to reference measurements. The large time delays between WhyCon samples are probably caused by a partial occlusion during the experiment. The average error is 1.4°. The low density of measurements makes the comparison of angular velocities impossible.

Figure 7.5 demonstrates the angular correction. An optical flow induced by the camera rotating around its x-axis is compensated using the angular velocity measured by the gyroscope.

Trajectory	$\mu\,[\mathrm{m}\cdot\mathrm{s}^{-1}]$	$\sigma\,[m\cdot s^{-1}]$
"Squares"	0.050	0.044
"Circles"	0.042	0.036

Table 7.1: Measured velocity errors

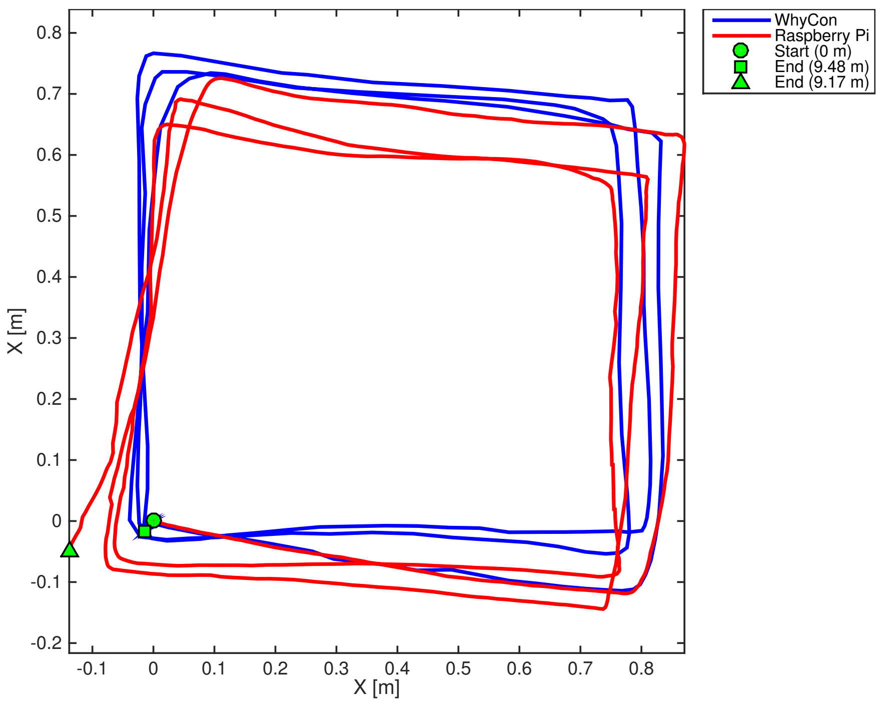

Figure 7.2: The "Squares" trajectory compared to WhyCon

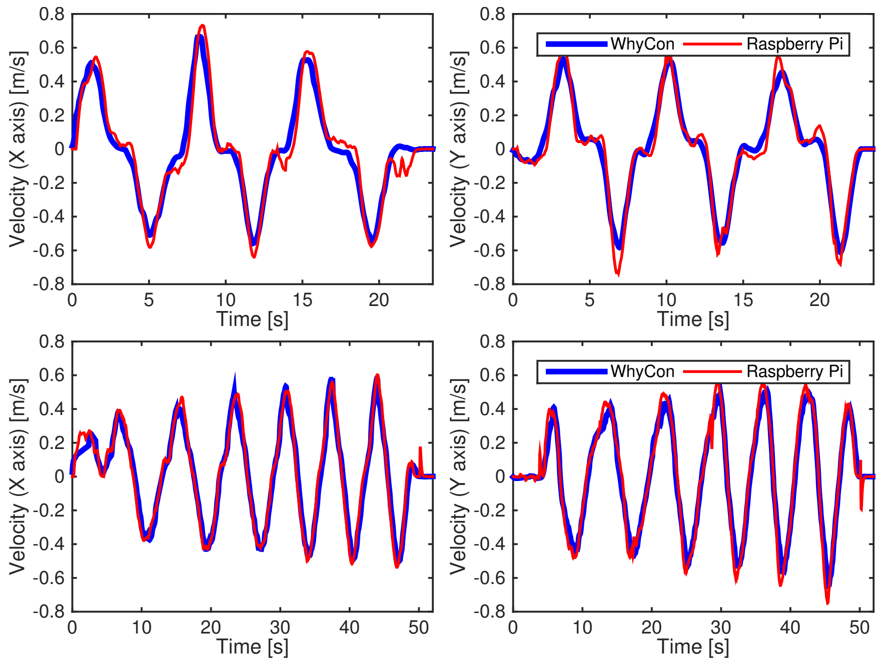

Figure 7.3: Measured velocity for trajectories "Squares" (top) and "Circles" (bottom) compared to WhyCon

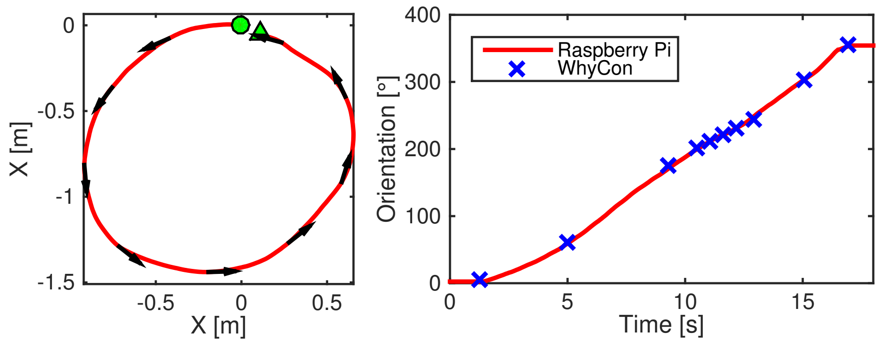

Figure 7.4: Computed rotation compared to WhyCon

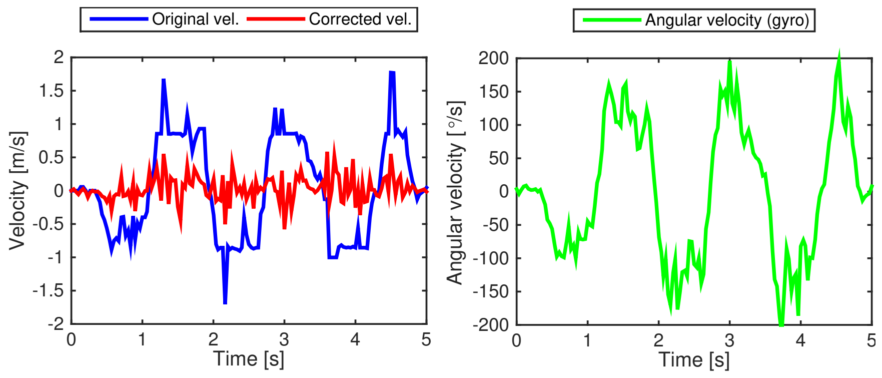

Figure 7.5: Demonstration of the angular correction

7.1.2 Processing time

Figure 7.6 shows the frame processing time required by the implemented solutions. Average processing times are listed in table 7.2. The mixed CPU and GPU solution is clearly the fastest one and can be even used with higher frame rates.

The approximately constant processing is caused by the fixed number of RANSAC iterations. This could be further lowered by using an adaptive method described in [HZ03].

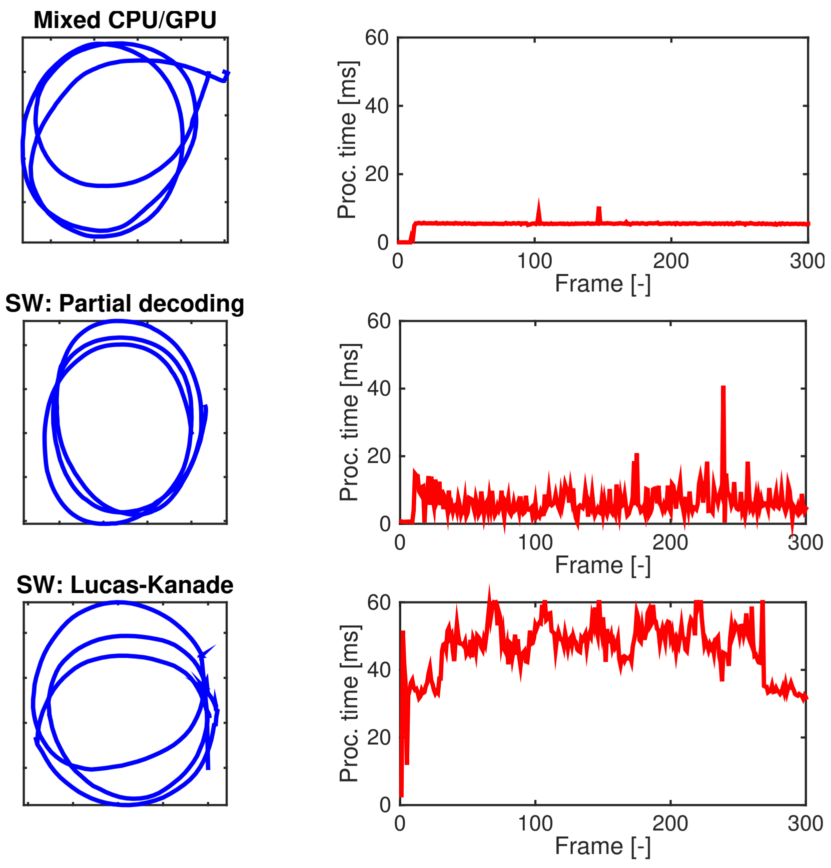

Figure 7.6: Comparison of processing times: Mixed CPU and GPU solution (top), Partial bitstream decoding (middle) and Lucas-Kanade (bottom)

Algorithm	Mixed CPU/GPU	Partial decoding	Lucas-Kanade
Processing time	5.36 ms	6.30 ms	46.43 ms

Table 7.2: Average frame processing times

7.2 Outdoor testing

Outdoor tests have been performed in a park environment using commercially available hexacopter DJI F550 (depicted on figure 7.7) equipped with a GPS unit and PX4Flow sensor for comparison. The hexacopter was remotely controlled by an operator to follow a given trajectory. Only the mixed CPU and GPU solution has been evaluated.

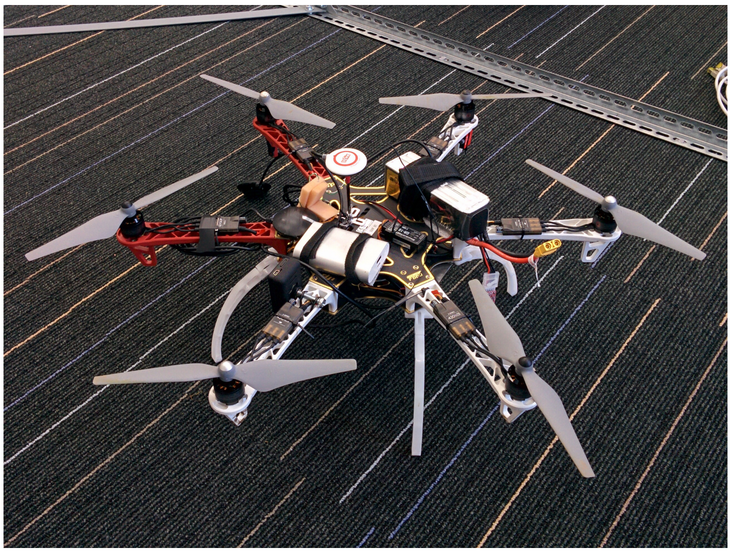
Figure 7.7: Hexacopter used for outdoor testing

7.2.1 Comparison with PX4Flow

As the PX4Flow sensor provides output in meters per second, the recorded velocity can be compared directly. It was however necessary to remove the ultrasonic sensor because it would cause interferences with PX4Flow's internal ultrasonic sensor. The altitude data from PX4Flow have been resampled and used as an altitude reference for the Raspberry Pi sensor. The orientation was ignored during this experiment in order to compare measurements with the PX4Flow sensor which does not recover orientation from the optical flow.

Figure 7.8 shows the trajectory integrated from Raspberry Pi measurements compared to the PX4Flow and figure 7.9 shows velocity comparison. The average difference is $7.56\,\mathrm{cm} \cdot \mathrm{s}^{-1}$.

Figure 7.10 shows a different trajectory recorded during a flight above pavement to demonstrate the ability to recover changes in orientation.

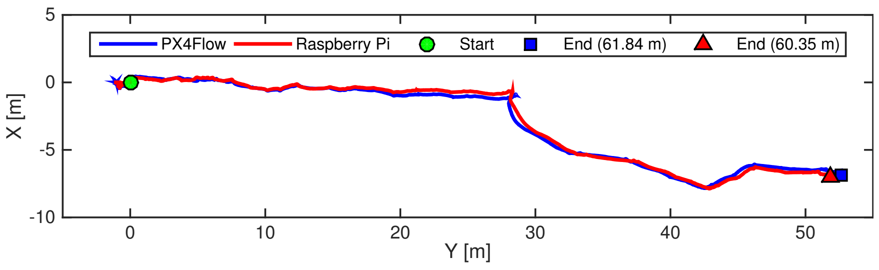

Figure 7.8: Computed trajectory compared to PX4Flow

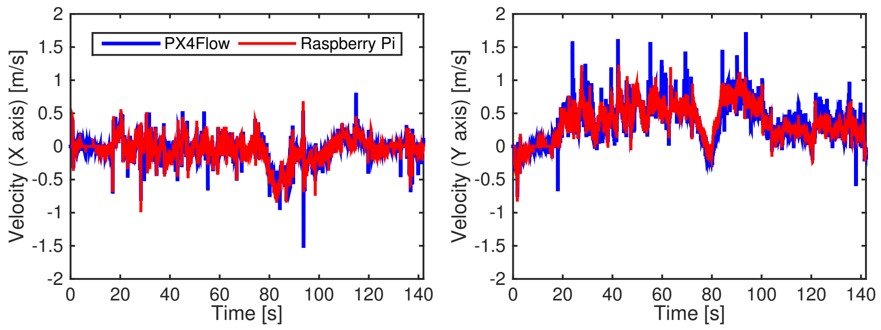

Figure 7.9: Computed velocity compared to PX4Flow

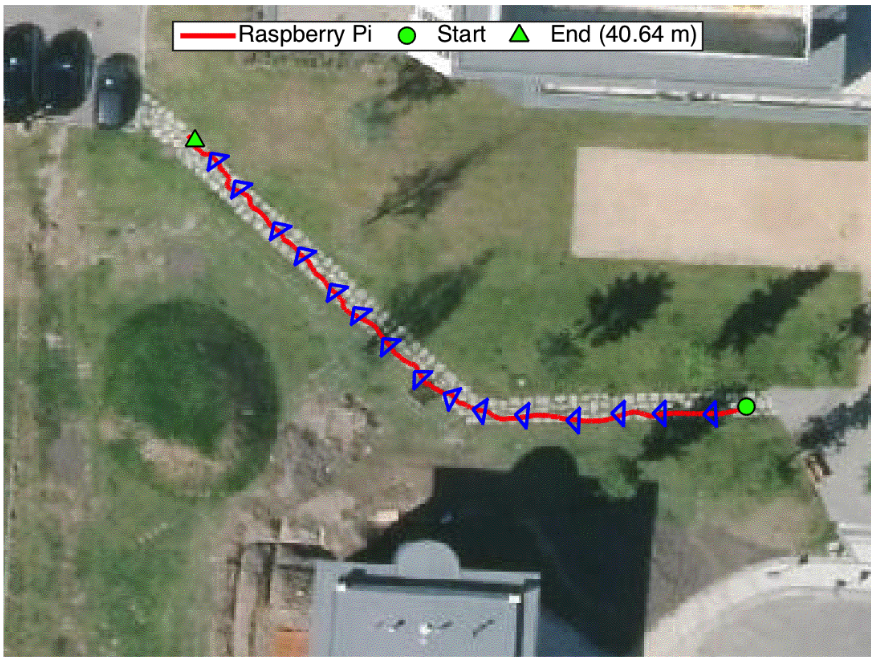

Figure 7.10: Integrated trajectory compared to map (map from [PC16])

7.2.2 Comparison with GPS

Figure 7.11 shows the position integrated by the Raspberry Pi compared to the position recorded from the GPS receiver. Geographic coordinates have been approximately converted to meters using the WGS 84 spheroid [Int03] and both trajectories have been aligned to have the same origin. The difference at the end of the 93-meter-long trajectory is approximately 0.7 meters.

Moreover, figure 7.12 shows the proportion of inliers identified by the RANSAC algorithm. The currently fixed number of iterations could be even lower as the average proportion of inliers is approximately 82%.

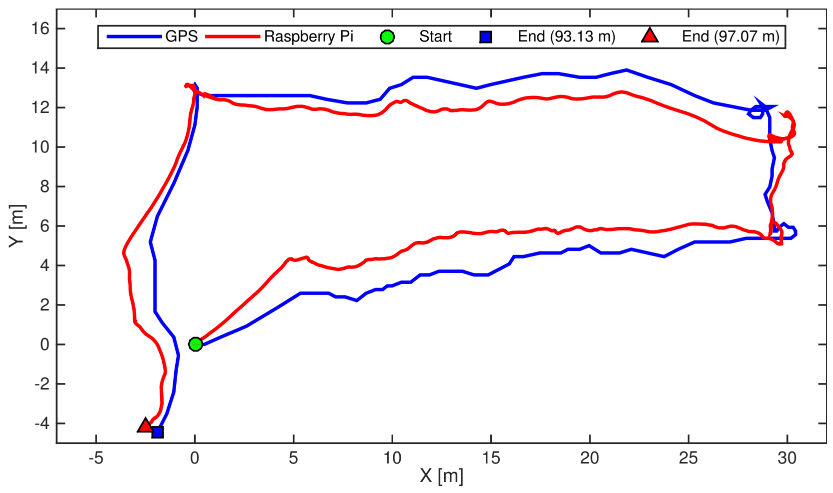

Figure 7.11: Integrated trajectory compared to GPS

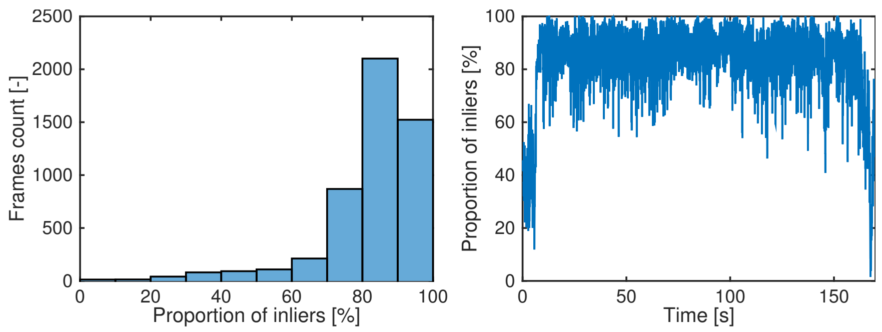

Figure 7.12: Proportion of inliers in time (for trajectory 7.11)

Chapter 8
Conclusion

I have designed and implemented a visual odometry sensor based on the hardware coarse motion estimation block featured by the Raspberry Pi single-board computer. I have also proposed alternative hardware-independent software solutions. The sensor is able to estimate the vehicle's velocity in both indoor and outdoor environments with a reasonable accuracy and provides an additional functionality when compared to the PX4Flow sensor. The focus on a popular and widely available single-board computer provides a wide range of options for potential future improvements. Moreover, I have contributed a few patches to the official RaspiVid application[1] and added a new functionality to it, so that a part of my work can be re-used in different scenarios.

The implemented solution runs in real-time. As the average processing time is under 6 milliseconds per frame, the frame rate can be increased from the current 30 frames per second to a higher rate of up to 90 FPS, which is the maximum supported by the Raspberry Pi.

While the proposed solution adds an additional functionality (orientation estimation) when compared to the PX4Flow sensor, it does not provide any configuration options. It has also not been tested as a feedback sensor for an actual flight controller. Potential future improvements may include a more efficient implementation of the gyro reading (featuring the internal FIFO and interrupt signals), filtering of the measured data or compensation for the effects of the camera's rolling shutter, as described in [KJBL11]. Future improvements might also include a custom PCB design which would replace the current prototyping board.

[1]https://github.com/raspberrypi/userland/commits/master?author=adamheinrich

Appendix A

Bibliography

[Ard16] ArduPilot. Mouse-based optical flow sensor (adns3080). *Copter documentation*, 2016. [Online; accessed 28 December 2016]. URL: `http://ardupilot.org/copter/docs/common-mouse-based-optical-flow-sensor-adns3080.html`.

[Ava08] Avago Technologies. *ADNS-3080. High-Performance Optical Mouse Sensor*, October 2008. [Datasheet].

[bfn14a] 6by9 (forum nickname). Re: Cme -x postponed? *Raspberry Pi*, 05 2014. [Online; accessed 23 December 2016]. URL: `https://www.raspberrypi.org/forums/viewtopic.php?f=43&t=76845`.

[bfn14b] 6by9 (forum nickname). Re: h264 encoding options? i want tune=zerolatency. *Raspberry Pi*, 04 2014. [Online; accessed 23 December 2016]. URL: `https://www.raspberrypi.org/forums/viewtopic.php?p=540152#p540152`.

[bfn15] 6by9 (forum nickname). Re: Fine motion estimation with picamera? *Raspberry Pi*, 01 2015. [Online; accessed 23 December 2016]. URL: `https://www.raspberrypi.org/forums/viewtopic.php?p=681715#p681715`.

[Bou01] Jean-Yves Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[Bra00] G. Bradski. Opencv. *Dr. Dobb's Journal of Software Tools*, 2000.

[BZF13] Adrien Briod, Jean-Christophe Zufferey, and Dario Floreano. Optic-flow based control of a 46g quadrotor. In *Workshop on Vision-based Closed-Loop Control and Navigation of Micro Helicopters in GPS-denied Environments, IROS 2013*, number EPFL-CONF-189879, 2013. URL: `https://infoscience.epfl.ch/record/189879/files/Optic-Flow_based_control_of_a_46g_quadrotor.pdf`.

[dev16] FFmpeg developers. Ffmpeg: A complete, cross-platform solution to record, convert and stream audio and video. https://ffmpeg.org/, 2016. [Software library].

[eLi16] eLinux.org. Rpi camera module, June 2016. [Online; accessed 28 December 2016]. URL: http://elinux.org/Rpi_Camera_Module#Technical_Parameters_.28v.2_board.29.

[FB81] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[Gá11] Bernát Gábor. Camera calibration with opencv, August 2011. [Online; accessed 7 January 2017]. URL: http://docs.opencv.org/master/d4/d94/tutorial_camera_calibration.html.

[HMTP13] Dominik Honegger, Lorenz Meier, Petri Tanskanen, and Marc Pollefeys. An open source and open hardware embedded metric optical flow CMOS camera for indoor and outdoor applications. In *2013 IEEE International Conference on Robotics and Automation*. Institute of Electrical and Electronics Engineers (IEEE), may 2013. URL: http://dx.doi.org/10.1109/ICRA.2013.6630805, doi:10.1109/icra.2013.6630805.

[Hol14] Gordon Hollingworth. Re: Cme -x postponed? *Raspberry Pi*, 05 2014. [Online; accessed 23 December 2016]. URL: https://www.raspberrypi.org/forums/viewtopic.php?p=548816#p548816.

[HS81] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial intelligence*, 17(1-3):185–203, 1981.

[Hug13] James Hughes. Raspivid. https://github.com/raspberrypi/userland/tree/master/host_applications/linux/apps/raspicam, 2013. [Computer software].

[HZ03] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Pr., 2003. URL: http://www.ebook.de/de/product/3267382/richard_hartley_andrew_zisserman_multiple_view_geometry_in_computer_vision.html.

[Int03] International Hydrographic Bureau. *User's Handbook on Datum Transformations Involving WGS 84*, 3rd edition, July 2003.

[Its15] Itseez. Open source computer vision library. https://github.com/itseez/opencv, 2015. [Software library].

[Jon13] Dave Jones. Picamera. a pure python interface to the raspberry pi camera module. `https://github.com/waveform80/picamera`, 2013. [Software library].

[Jon16] Dave Jones. Camera hardware - picamera 1.12 documentation, June 2016. [Online; accessed 7 January 2017]. URL: `http://picamera.readthedocs.io/en/release-1.12/fov.html`.

[KB17] Jonghyuk Kim and Galen Brambley. Dual optic-flow integrated inertial navigation for small-scale flying robots. In *Australasian Conference on Robotics and Automation*, 2017. URL: `http://www.araa.asn.au/acra/acra2007/papers/paper181final.pdf`.

[KG11] Tim Kazik and Ali Haydar Goktogan. Visual odometry based on the fourier-mellin transform for a rover using a monocular ground-facing camera. In *2011 IEEE International Conference on Mechatronics*. Institute of Electrical and Electronics Engineers (IEEE), apr 2011. URL: `http://dx.doi.org/10.1109/ICMECH.2011.5971331`, `doi:10.1109/icmech.2011.5971331`.

[KJBL11] Alexandre Karpenko, David Jacobs, Jongmin Baek, and Marc Levoy. Digital video stabilization and rolling shutter correction using gyroscopes. *CSTR*, 1:2, 2011.

[KNP$^+$12] Tomas Krajnik, Matias Nitsche, Sol Pedre, Libor Preucil, and Marta E. Mejail. A simple visual navigation system for an UAV. In *International Multi-Conference on Systems, Sygnals & Devices*. Institute of Electrical and Electronics Engineers (IEEE), mar 2012. URL: `http://dx.doi.org/10.1109/SSD.2012.6198031`, `doi:10.1109/ssd.2012.6198031`.

[LK$^+$81] Bruce D Lucas, Takeo Kanade, et al. An iterative image registration technique with an application to stereo vision. In *IJCAI*, volume 81, pages 674–679, 1981.

[Max14] MaxBotix. *HRLV-MaxSonar-EZ Series. High Resolution, Precision, Low Voltage Ultrasonic Range Finder*, 2014. [Datasheet].

[Mei09] Lorenz Meier. Mavlink protocol c/c++ implementation. `https://github.com/mavlink/c_library_v1`, 2009. [Software library].

[PC16] IPR PRaha and CUZK. Praha. `http://www.geoportalpraha.cz/mapy-online`, 2016. [Online map; accessed 20 December 2016].

[Piv16] Tomáš Pivoňka. Vizuální lokalizace pro experimentaci v mobilní robotice (motion capture system for experimentation in mobile robotics). Master's thesis, Czech technical university in Prague, 2016.

[Ric10] Iain Richardson. *H.264 Advanced Video Compression Standard*. Wiley-Blackwell, 2010. URL: http://www.ebook.de/de/product/8759262/iain_richardson_h_264_advanced_video_compression_standard.html.

[Sal13] Rafael Muñoz Salinas. Raspicam: C++ api for using raspberry camera with/without opencv. http://www.uco.es/investiga/grupos/ava/node/40, 2013. [Software library].

[Sch02] Adam Schneider. Gps visualizer. http://www.gpsvisualizer.com/, 2002. [Online tool; accessed 5 December 2016].

[SCN13] Kathryn Schneider, Joseph Conroy, and William Nothwang. Computing optic flow with ardueye vision sensor. Technical Report ARL-TR-6292, Army Research Laboratory, January 2013. URL: http://www.dtic.mil/get-tr-doc/pdf?AD=ADA572633.

[STM13] STMicroelectronics. *L3GD20H. MEMS motion sensor: three-axis digital output gyroscope*, March 2013. [Datasheet].

[TG12] Mike Thompson and Peter Green. Raspbian, 2012. [Computer software].

[Upt14] Liz Upton. Vectors from coarse motion estimation. *Raspberry Pi*, 04 2014. [Online; accessed 23 December 2016]. URL: https://www.raspberrypi.org/blog/vectors-from-coarse-motion-estimation/.

[WC11] Andreas Wedel and Daniel Cremers. *Stereo Scene Flow for 3D Motion Analysis*. Springer-Verlag GmbH, 2011. URL: http://www.ebook.de/de/product/15361410/andreas_wedel_daniel_cremers_stereo_scene_flow_for_3d_motion_analysis.html.

[YSK09] Wonsang You, M. S. Houari Sabirin, and Munchurl Kim. Real-time detection and tracking of multiple objects with partial decoding in h.264/AVC bitstream domain. In Nasser Kehtarnavaz and Matthias F. Carlsohn, editors, *Real-Time Image and Video Processing 2009*. SPIE-Intl Soc Optical Eng, feb 2009. URL: http://dx.doi.org/10.1117/12.805596, doi:10.1117/12.805596.

Appendix B

Attached CD

The attached CD contains all developed programs and the text of this thesis.