Quadrocopter Stabilization using Neuromorphic Embedded Dynamic Vision Sensors (eDVS)

eingereichte
PROJEKTPRAKTIKUMSARBEIT
von

Andreas Pflaum,Franz Eutermoser

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss
Univ.-Prof. Dr.-Ing. Sandra Hirche

Betreuer:	Cristian Axenie
Beginn:	23.04.2012
Zwischenbericht:	11.06.2012
Abgabe:	29.06.2012

Inhaltsverzeichnis

Kapitel 1

Introduction (Andreas)

This report of the practical course *Quadrocopter stabilization using neuromorphic Embedded Dynamic Vision Sensors (eDVS)* is a description of the components, steps and goals reached during its time period.

1.1 Motivation

Currently several industries and research facilities work on unmanned aerial vehicles (UAVs) for military, security or research applications. Quadrocopters are a common solution, when a stationary and stable system is required, as it is used for monitoring or operations in crowded or narrow environments. Besides as light as possible components, the stability is the most crucial aspect.

1.2 Task description

Therefore in this project it shall be tested, if the common sensors used for stabilization like IR or sonars on the quadrocopter can be replaced by the recently developed lighter, faster and less resources lasting neuromorphic Embedded Dynamic Vision Sensors. Due to the different sensors and closed software system of the drone, new sensor processing and controller for stabilization must be implemented.

The task is to fly the quadrocopter with the onboard sensors and controller in a stable position and from then on to use the new sensors and controller to stabilize the vehicle from disturbances in the pitch and roll direction.

1.3 Approach

Two of the eDVS will be used and fixed on the quadrocopter to sense changes in the environment. These events are processed with an optical flow algorithm and finally used to compute the rotational displacement of the vehicle. These points are explained in chapter 3.

The displacement is then used by the developed controller to move the quadrocopter back to the stable orientation and is described in the chapter thereafter.

Kapitel 2

Control of the Quadrocopter AR.Drone (Franz)

2.1 Quadrocopter *AR.Drone*

A quadrocopter is a multicopter, which is lifted and propelled by four rotors. Quadrocopters work similar to helicopters, but they do not need any tail rotor, because each pair of opposite rotors is turning the same way - one pair is turning clockwise and the other anti-clockwise.
In this section, we introduce the quadrocopter AR.Drone of Parrot. Afterwards, we will introduce the software development kid (SDK) or application programming interface (API).

2.1.1 AR.Drone

The AR.Drone of Parrot is a radio controlled flying quadrocopter [Parb] and [Para]. It is illustrated in Fig. 2.1. It can be controlled with any Computer, and many other electronic devices. Communications are through WiFi and USB. The structure is contructed of carbon-fibre tubes. The electric motors are brushless type. They supply 15 watt. The AR.Drone is configurated as a self-stabilizing system. The inertial measurement unit uses a MEMS 3-axis accelerometer, 2-axis gyroscope and a single-axis yaw precision gyrometer. With assistance of an ultrasonic altimeter, with a range of 6 meters, the quadrocopter stabilizes the height. Two cameras are mounted on the drone. One shows on the floor, the other forwards.

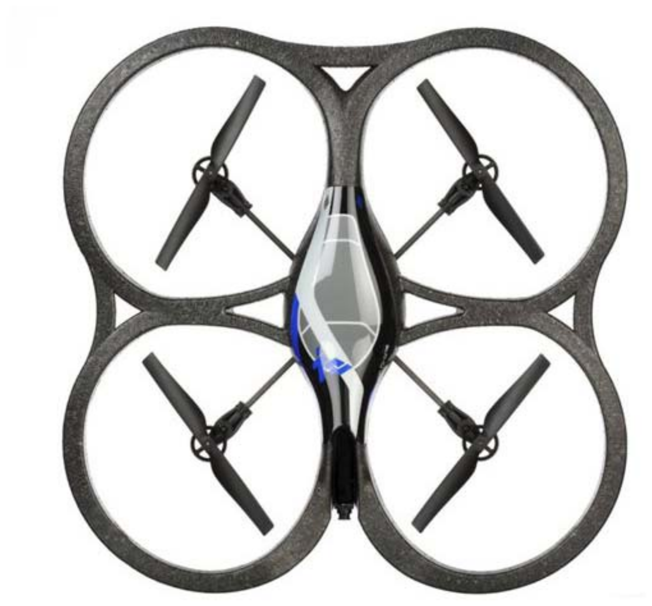


Figure 2.1: Ar.Drone of Parrot [Parb]

The structure is constructed of carbon-fibre tubes. There exist two configurations, the indoor and outdoor configurations. Because of safety reasons, especially the rotors have to be protected indoors. This configuration is illustrated in Fig. 2.1. It has a weight of 420 g. The original battery pack ensures a flight of about 12 minutes. The outdoor configuration is illustrated in 2.2

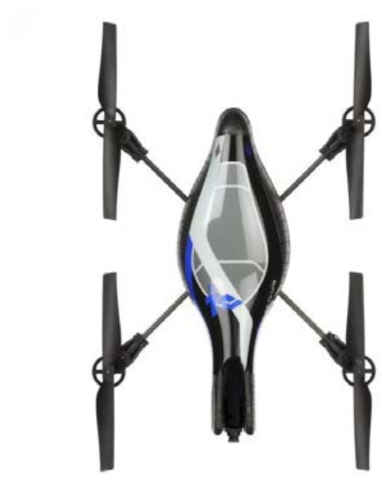


Figure 2.2: AR.Drone without indoor safety hull [Parb]

To adjust the existing control application, we use a the SDK of Parrot.

2.1.2 SDK/API - Functionality

We control the AR.Drone with aid of a Linux computer via WIFI. The main control algorithms are computed on-board with a ARM9 microcontroller with 468 MHz. It has a embedded Linux operating system. The original programm for the linux computer has only the task to provide the on-board controller with user-data and to receive data. The only way to control the drone with the computer is to command basic maneuvres. These are take-off, hovering with constant altitude, landing and desired inclinations and height.
Fig. 2.3 shows the layered architecture of a host application.

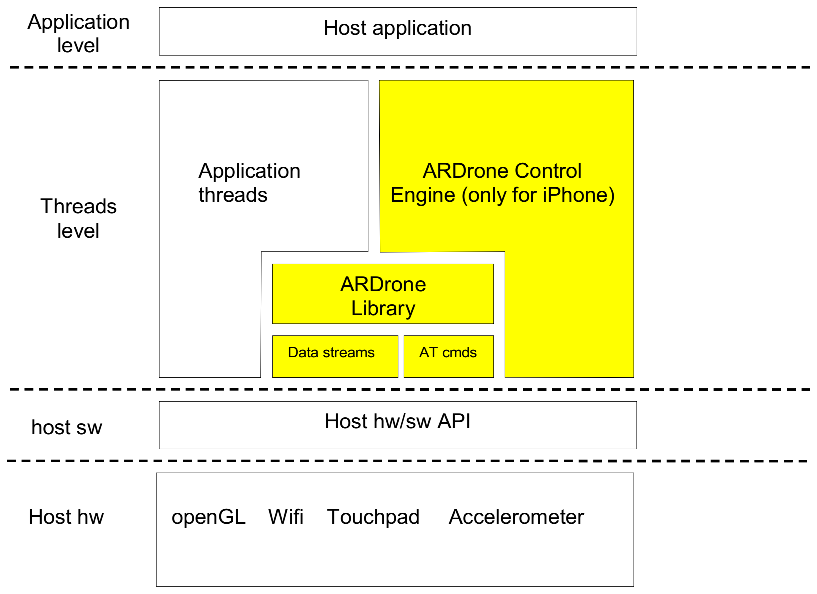


Figure 2.3: Layered architecture of a host application [Para]

The AR.Drone Library is provided as an open-source library with high level APIs to access the drone. One of its parts is the ARDroneTool. The ArDroneTool provides [Para]:

- an AT command management thread, which handles the communication to the drone
- a navdata management thread, which receives data from the drone, decodes it to provide the client application with data

- a video management thread, which handles the video data
- a control thread, which controls the communication with the drone

These threads take care of the connection to the drone. They only run in case of a stable connection. The vp_com library is responsible for reconnecting to the drone when necessary.
The basic command for control of the drone is ardrone_at_set_progress_cmd. The input of that function is the roll- and pitch-angle, the vertical speed and the yaw angular speed.

2.2 Controller Design

The goal of this section is to describe all theoretical aspects in respect to the controller design. In the first subsection, we deal with the basic structure of the project. In the second, we treat the identification of the system and build a mathematical model. The last point of this section is the design of the controller.

2.2.1 Control Structure

In this subsection, we will deal with the basic structure of the system. Our initial task was to stabilize the drone in the roll and pitch axis. This means to control the drone, so that it keeps horizontally. Any inclination of the drone will lead immidiatelly to a translational movement, hence for our task, we only consider the roll and pitch angle.

The original abstract structure, concerning the roll and pitch angle, is a simple feedback loop with the on-board controller, the sensors and the drone. This is illustrated in Fig. 2.4.

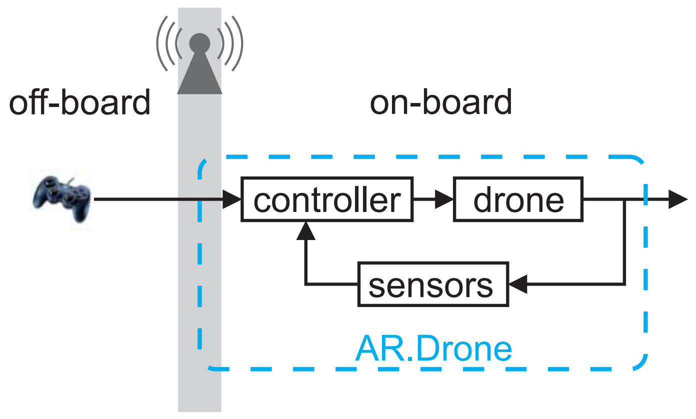


Figure 2.4: original control structure

For the stabilization task with the eDVS sensors, we need to separate the on-board controller with the original sensors from the drone. But as we can not do this with the SDK, it is not possible to do this with reasonable effort.
We changed the project task to create a cascaded control loop with the eDVS sensor and use the input as a disturbance entry. This is illustrated in Fig. 2.5.

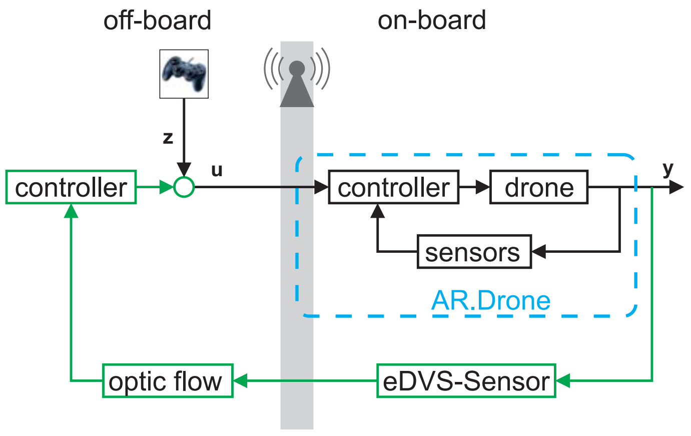


Figure 2.5: control structure of our new task

The effect of the changed task is, that we do not know the transfer function of the on-board processed part of the AR.Drone. Hence, every physical or aerodynamical determination is senseless and we have to fit a model into measured behavior of the drone.

2.2.2 System Identification

In this subsection, we determine a dynamical model of the on-board controlled drone. The approach is measurement based, that means we compare a certain system entry with measured systems reaction. We developed a test environment, where it is possible to command steps and where record measurements. One typical example of the recorded data is shown in Fig. 2.6. This illustrates both axes, whereas we only commanded on one axis several steps.

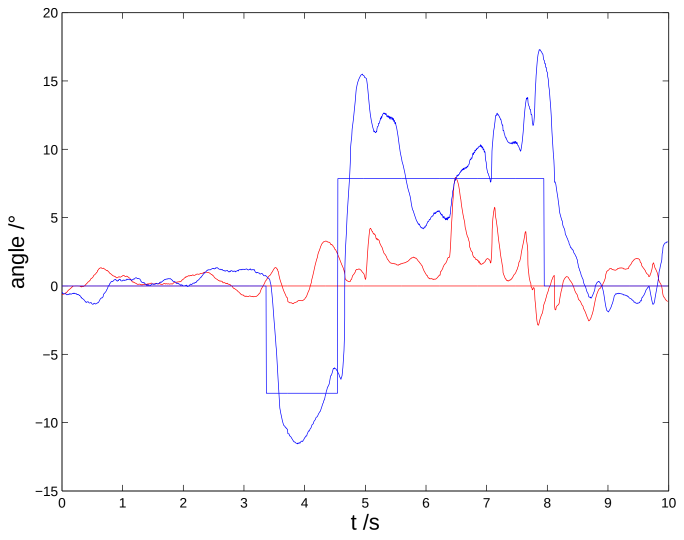


Figure 2.6: system behavior for commanded steps in both angles

This figure illustrates the system's high sensitivity on disturbances like circulation of air. Based on such step responses is it hard to determine any model. For that reason, we make several measurements and filter these data by overlaying all standardized steps. The resulting steps are illustrated in Fig. 2.8. Due to the similar result for both axis, we assume the same behavior for both axis. The resulting step is illustrated in Fig. **??**.

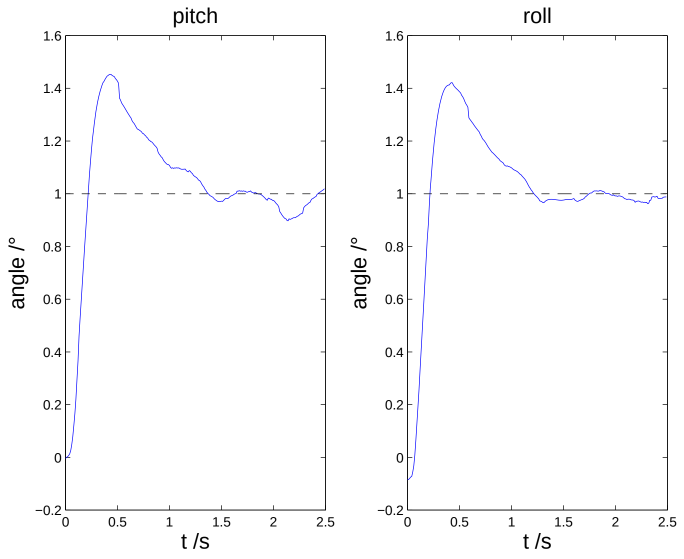


Figure 2.7: standardized filtered steps, left pitch axis, right roll axis

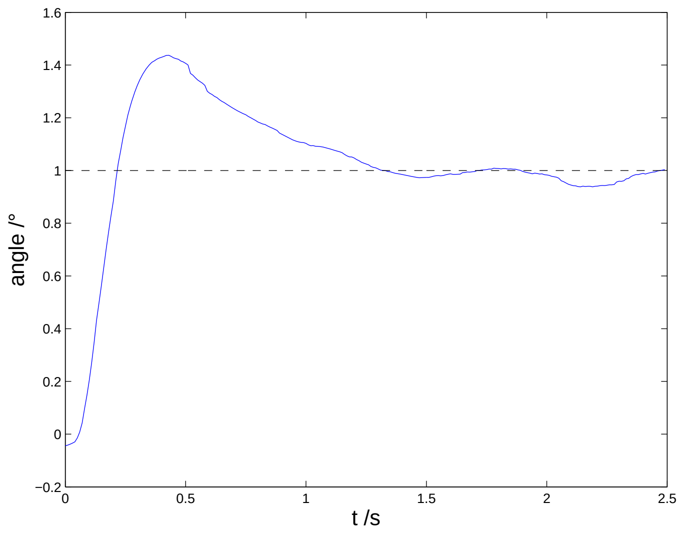


Figure 2.8: standardized filtered step

In order to fit a model into this step, we assume this to be a system third order with one zero. The summands, which do not depend on s have to be the same by regarding standardized steps / systems. Hence, we have five parameters to optimize.

$$G_{\text{sys}} = \frac{n_1 s + n_2}{p_1 s^3 + p_2 s^2 + p_3 s + n_2} \tag{2.1}$$

Our approach for fitting that system is a stochastical approach. This can be described as follows:

- determine random parameters near the parameters of the actual best system
- determine the system and the step response of the actual parameters
- compute the absolute error by integrating the quadratic difference of the measured and computed step
- memorize this system as the new best system, if this error is smaller than the error of the actual best system
- restart

This procedure converge fast to the following result:

$$G_{\text{sys}} = \frac{1136s + 1868}{10.81s^3 + 147.9s^2 + 955.8s + 1868} \tag{2.2}$$

The step response of that system is illustrated in Fig. 2.9

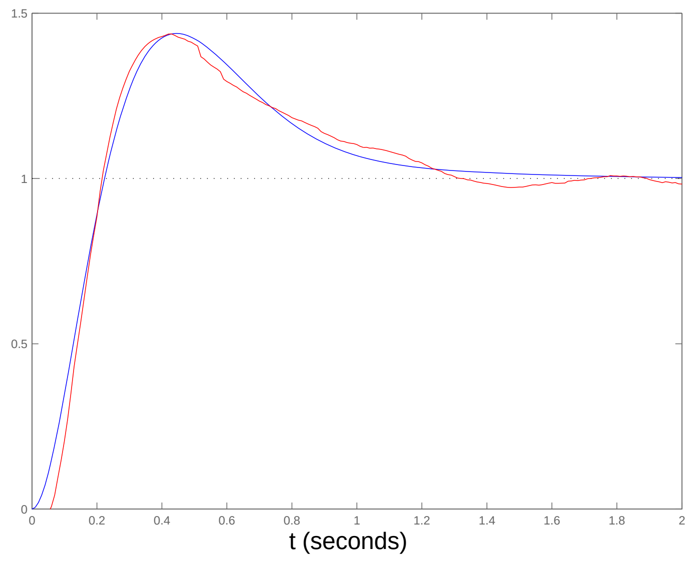


Figure 2.9: simulated and measured step

2.2.3 Controller

A perfect controller is in our system not possible, because of several reasons. First, we have not enough knowledge of our system. Second, the hardware has to stay adjustable, because of different masses of batteries for example. Because of these reasons, we decided to use a very simple rule of thumb of [ZR11], to get a first idea of the parameters. Afterwards, we will adjust these parameters manually, till the system reaches our favored behavior. In our case, we get from the model a good PID controller with the following parameters:

$$K_P = 2.0 \text{ - } K_I = 2.0 \text{ - } K_D = 0.1 \tag{2.3}$$

The resulting step response is illustrated in Fig. 2.10 and the poles and zeros in Fig. 2.11.

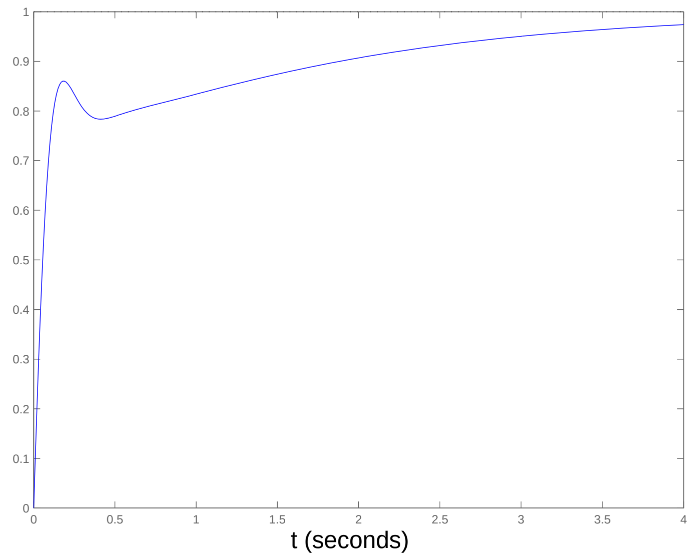


Figure 2.10: step response of the controlled system

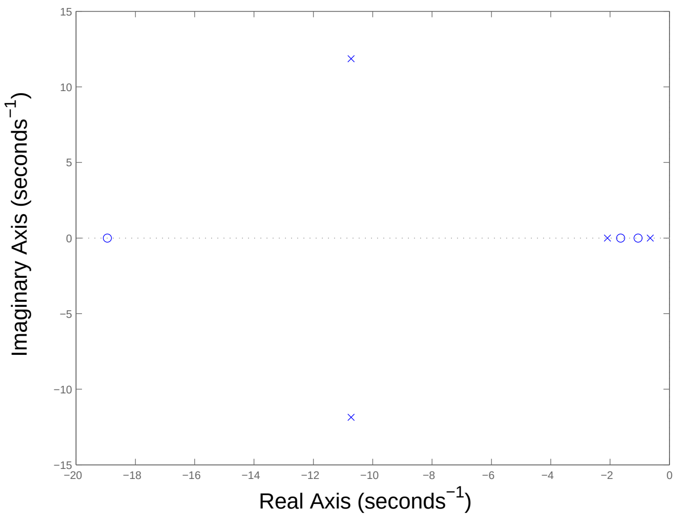


Figure 2.11: poles and zeros of controlled system

The interesting aspect of the poles and zeros is that there are very slow poles, but close to them zeros. Hence they are almost reduced itself.

2.3 Realization of the Control Part

In this chapter, we will describe our programs. For the basic functionality, we refer to the section of the SDK/API functionality.
In our program, we mainly use two threads for the controller. The first is the navdata

management thread, the other is the at command management thread. Both communicate with each other. We explain the basic functionality with the functionality of the gamepad. This is illustrated in Fig. 2.12.

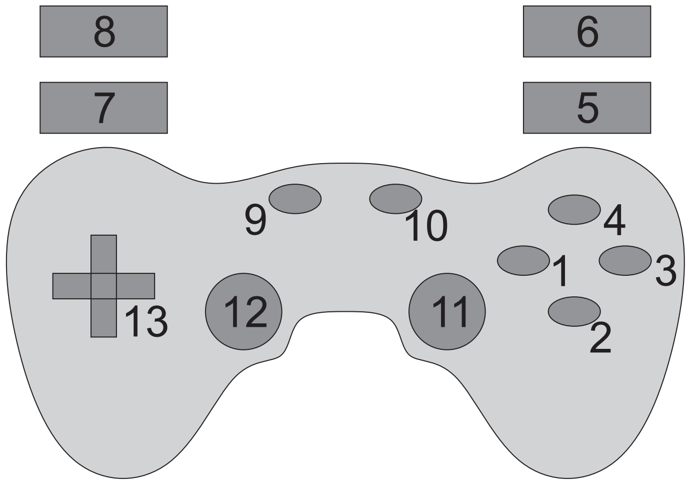


Figure 2.12: gamepad - numbers refer to functionality

- (1) - start recording
- (2) - stop recording
- (3) - test mode on
- (4) - test mode off / reset
- (5) - standard control
- (6) - set zero
- (7) - no more used
- (8) - control mode
- (9) - activate drone
- (10) - start/land drone
- (11) - control lever
- (12) - trim drone
- (13) - steps (only in test mode active)

The first step for programming the controller was to adjust the existing code, that we could use the available gamepad. Afterwards, we had to program the test mode, in order to be able to record data, especially the actual and desired roll and pitch angles. In test mode, the standard control lever is deactivated, but it is possible to apply steps to the drone. With the set zero and the trim functionality, it is possible to reduce the drifting of the drone. In the control mode, the controller is active, the control lever is interpreted as the disturbance entrance of Fig. 2.5.
In conclusion, we could create a controller, which stabilizes the drone. We could not test the whole system yet, so we had to do some tests with the standard sensors, instead of the eDVS sensors. The result is, that the system is very robust against disturbances, like our manual disturbance entrance. Besides, changes of more than 30% of the mass did not have big negative influence.

Kapitel 3

Sensor and data acquisition (Andreas)

As mentioned in the task description, a neuromorphic sensor and the optical flow algorithm is used in this project to get the displacement of the quadrocopter for stabilization. This will be worked on in this chapter.

In the first section the sensor will be introduced, its interface explained and how the sensor data are preprocessed for further steps. The second part describes the optical flow algorithm implemented in this system and how the output is used to compute the rotational displacement of the quadrocopter.

3.1 Neuromorphic sensor *eDVS128*

The dynamic vision sensor (DVS) *eDVS128* is an event-based sensor and therefore more efficient using resources than a frame-based sensor sending all data at a given time interval. Fig. 3.1 shows one of the neuromorphic sensors.

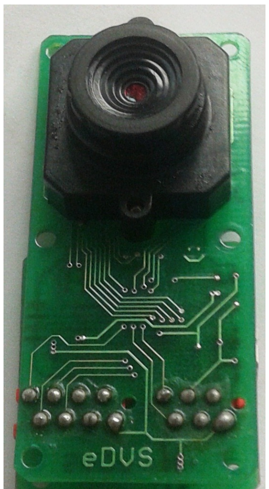


Figure 3.1: Picture of the neuromorphic event-based sensor eDVS128

The word *neuromorphic* describes the inspiration of this retina vision sensor by the retina in neurobiology and the asynchronous signal behavior. The eDVS128 used in this project has a sensor matrix of 128x128 pixels. Every quantized change of log light intensity at each individual pixel goes into two comparators that decide if there is a brightness change over a threshold or not. When the sensor detects a change at a pixel, this as an event is stored with other pixel events within the same time range in a queue and send to the serial interface. Every event consists of the X-index and Y-index of the pixel, a parity bit to know if the light is now brighter or darker than before and a timestamp in μs, when exactly it was detected. Fig. 3.2 displays such events visualized in a matrix when moving a hand in front of the sensor. The black areas represent no data, whereas the red and green pixels represent a de-/increased brightness.

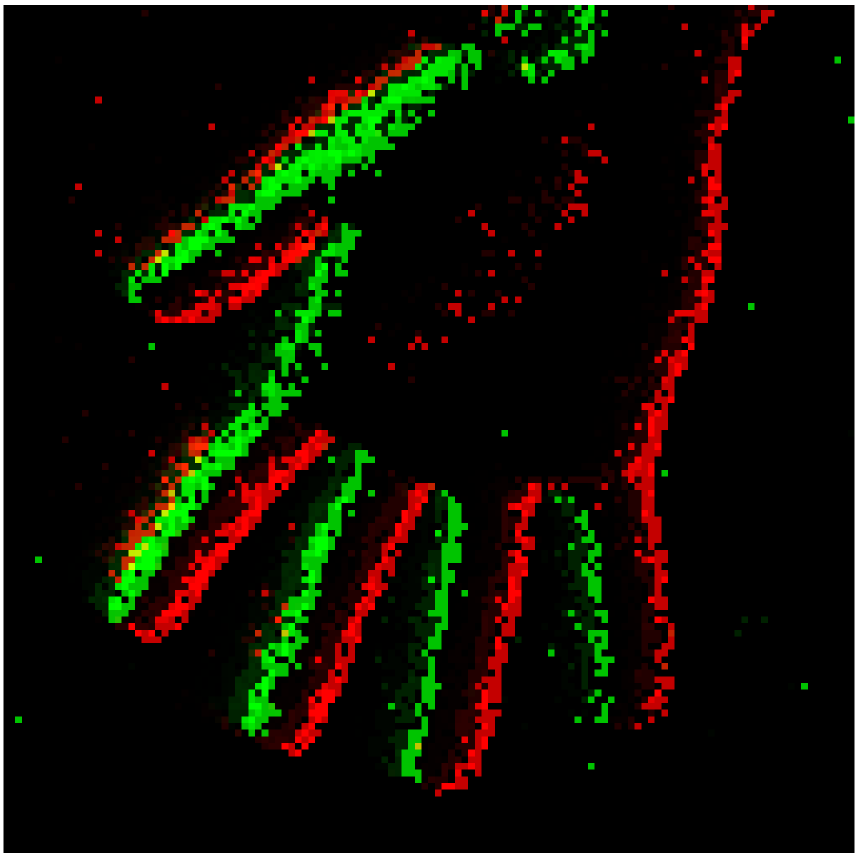
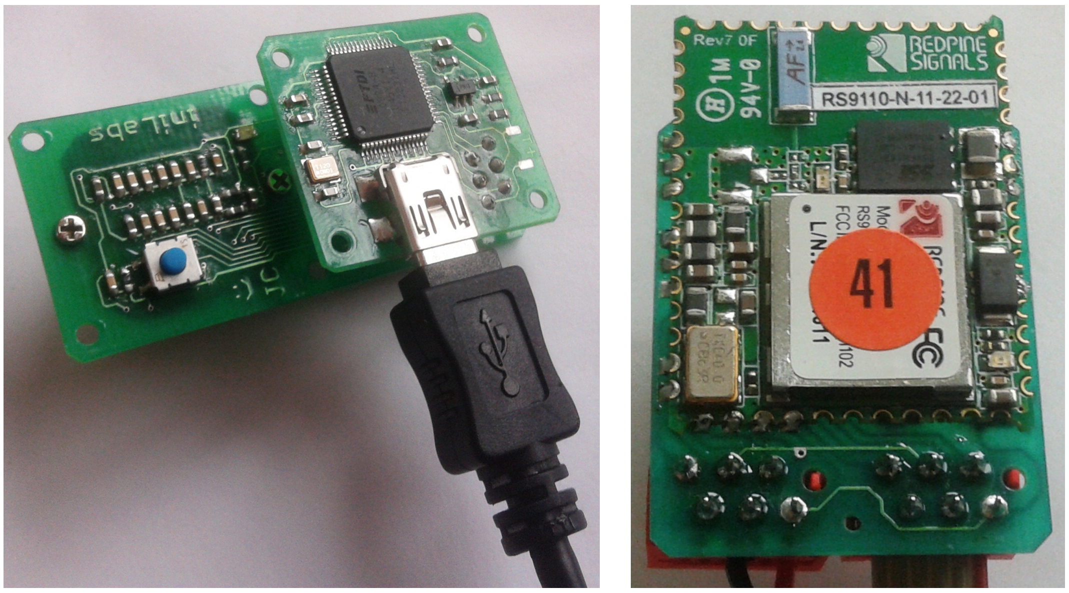


Figure 3.2: Example of events received from the sensor (black: no data, red/green: event of de-/increased brightness)

It can be seen that in contrast to frame based sensors, where the complete frame is send at a given rate even if nothing has changed, the eDVS128 sends only data when there was a contrast change and only the data that changed. Fig. 3.2 shows that only a small part of the pixels contain events (red/green) and therefore how much bandwidth can be saved by this event-based sensor (black areas). As a result much less data need to be transmitted and processed, which can be exploited by smaller and lighter components or faster execution. This is an important benefit for light weight vehicles such as the quadrocopter.

3.1.1 Connection interface

The first task in this project after reading into the features of the eDVS128 was to build up a connection between the sensors and the pc used for the computations, which is explained now. To get the events from the serial interface on the sensor to the PC, USB adapters (Fig. 3.3(a)) and WLAN adapters (Fig. 3.3(b)) are available.



(a) FTDI USB adapter attached to the eDVS128 (b) Redpine WLAN adapter

Figure 3.3: USB and WLAN adapter for the eDVS128

To access the data on the PC, java, C++ and Matlab templates can be used as well as a terminal test program for Windows[1]. Since the control part and part for the AR. Drone are written in C++, it is also used for the sensor part of the project.

USB adapter

To have more reliable data at the beginning as with slower and lossy WLAN connections, the FTDI USB adapter (Fig. 3.3(a)) was implemented first based on the C++ template. Unfortunately the first used eDVS128 sensor was not working correct. The test programs worked only on some 32-Bit Windows and Linux desktop PCs, but also not reliable, whereas on 64-Bit systems and tested 32-Bit Windows and Linux

[1]https://wiki.lsr.ei.tum.de/nst/programming/edvsgettingstarted

operating systems on laptops it did not work with any driver changes and deeper operating system settings. Most of the time, a connection could be established, but no data could be send to the sensor or received from it.

Operating systems that were tested: Windows 7 32/64-Bit, Windows XP 32-Bit, Linux 10.04 32-Bit and Linux 11.10 64-Bit.

Besides the C++ program, the java test program was tested with the Java Runtime Environment 6 and 7 and the newest serial port drivers[2], but also the Matlab programs and the terminal program were tested without success.

WLAN adapter

Since USB-cables from a flying quadrocopter cannot be used and a direct connection of the USB-adapters to the board of the quadrocopter would be too complex for this short project, the WLAN adapter (Fig. 3.3(b)) is used in this project. It is the *WLAN Module Redpine RS9110-N-11-22*[3] developed (as the eDVS128) at the Neuroscientific System Theory (NST) department of the TUM.

To avoid too much packet losses and because of problems with the use of UDP as the protocol in other projects, TCP is used. The adapter is registered with its MAC-address in the router of the used network and gets a static IP when switching on. Then it starts sending events over the port 56000 and everyone in the same network IP range can open a TCP socket with the data of the WLAN adapter and get the data.

Unfortunately also the first used Redpine WLAN adapter was not working correct. When there were too much events recognized at the same time, as it is when the quadrocopter moves, the sensor gets stuck and had to be power cycled. By extensive tests with different modules, some get stuck with less events and some with more events or almost never, it was shown, the first up to a few minutes they work fine until the voltage regulator gets very hot. As the weather at that time was about 30°C and the WLAN adapter in other project also make problems, the power supply was reduced from 5 V to 4.5 V (below 4.5 V there are a lot of data errors) and the voltage regulator was relieved by another one. But as this did not help, the code running on the module was analyzed and adapted by some timeouts concerning the communication between the microcontroller and WLAN chip on the adapter. With this adaption, the Redpine WLAN adapters now work without getting stuck and the new firmware is flashed on all the modules that make problems.

At the end, although the long and extensive tests with the eDVS128 sensor and the

[2]http://rxtx.qbang.org/wiki/index.php/Download
[3]https://wiki.lsr.ei.tum.de/nst/setup-redpine/index

USB and WLAN modules lead to a delay of this project, other projects using it now benefit from this work.

3.1.2 Data preprocessing

As mentioned in the last subsection, the program used here is based on the available C++ template for the USB connection. Therefore a TCP receiver was implemented to get the sensor data over WLAN to the PC. The GUI is programmed with the help of the Qt library (Fig. 3.13). For every TCP connection to a sensor a separate thread needs to be started and the data then send to the main thread including the GUI and computation parts. An overview of the program structure for the sensor processing can be seen in Fig. 3.4. The lower left shows that the computed displacements at the end are also send over TCP to the Controller, which runs on the same PC. For that a TCP sender thread was implemented and for receiving on the controller side the code of the TCP receiver was reused.

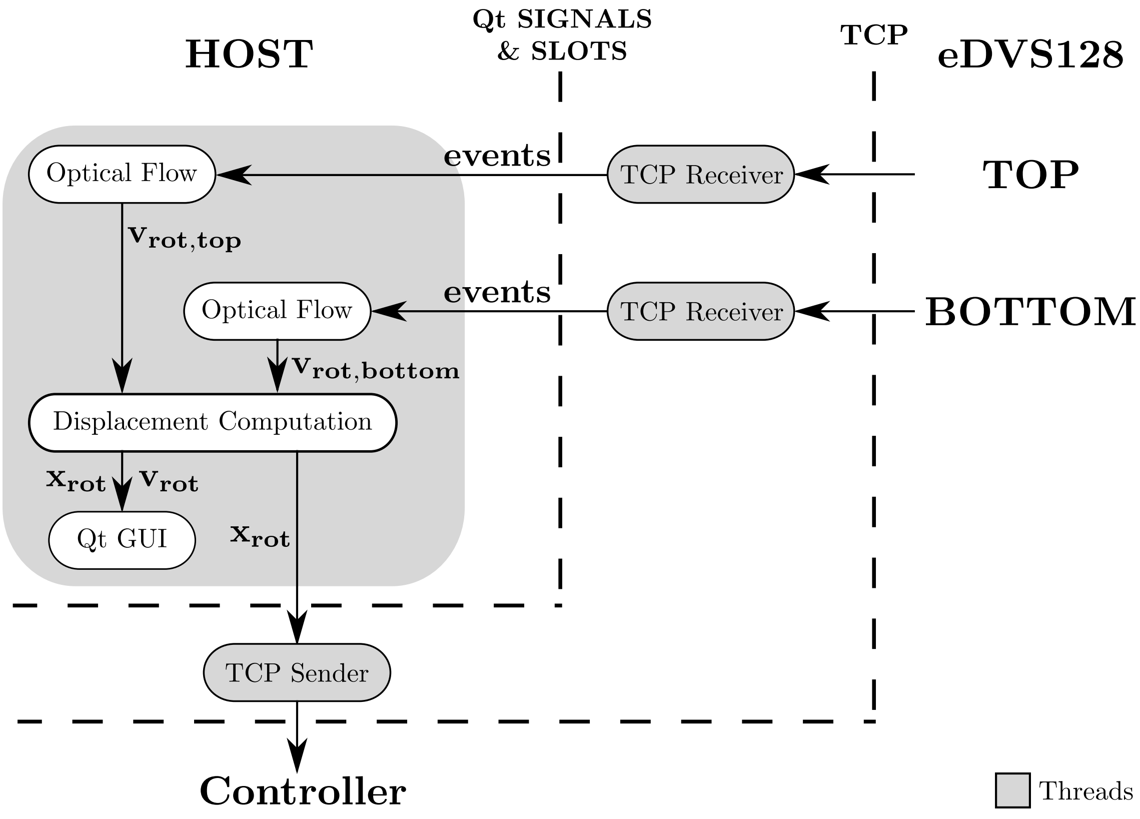


Figure 3.4: Program structure of the sensor processing

As it can be seen in the diagram, the optical flow algorithm needs and gets a sequence of events ordered by increasing timestamp as input.

One event consists normally of 32 Bit:

```
0XXXXXXX pYYYYYYY TTTTTTTT TTTTTTTT
```

The first Bit of every 32 Bit block must be zero followed by 7 Bits (128 pixels) for the pixel X-index. If the first Bit is not zero, as it sometimes happens because of the sensor switching the Y- and X-index, it is jumped to the next 32 Bit block. The next Byte begins with the parity Bit, followed by the 7 Bits (128 pixels) for the pixel Y-index. The last 16 Bit represent the timestamp of the event in μs. That means it can only reach 65535 μs and starts then from 0 μs again, as can be seen in Fig. 3.5.

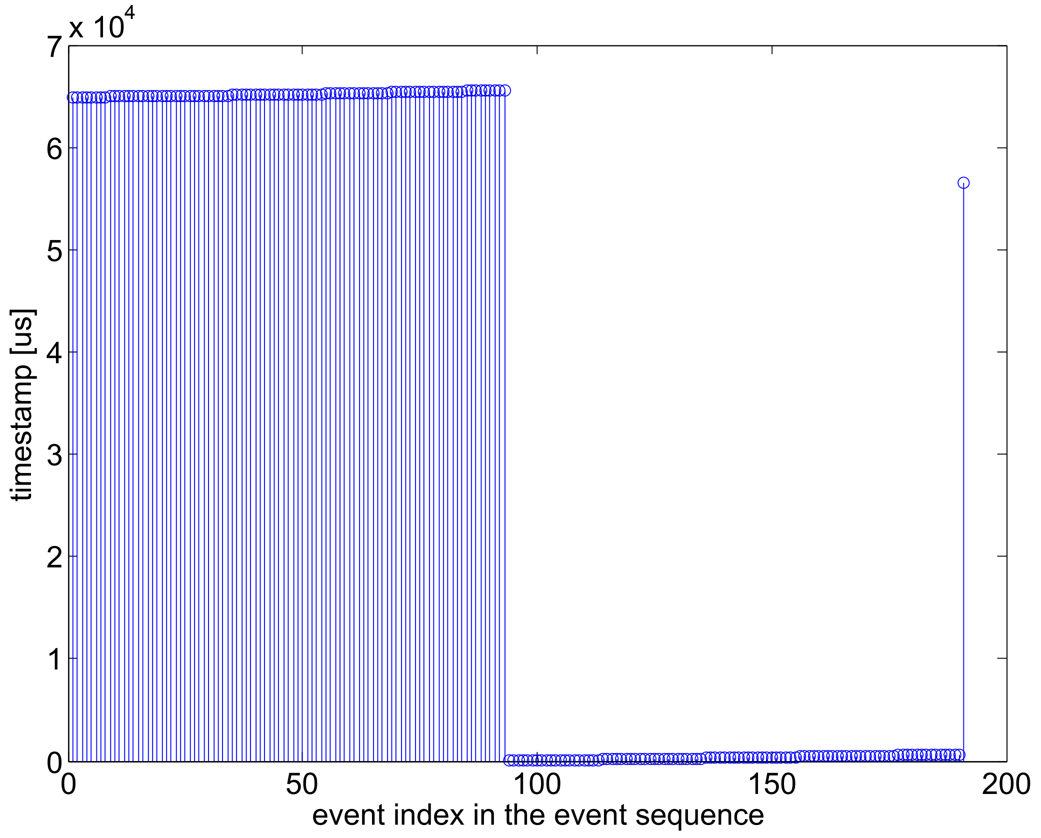


Figure 3.5: Timestamps of an event-set reaching the Maximum of 65535 μs

Therefore the next common step was to integrate the timestamps, which means to add every time an additional 65535 μs from the time on, the next timestamp is lower than the first one. The result was right for the sensor not detecting anything except for noise or small and slow motions. When there were too much events, some timestamps were corrupted, like the last timestamp in Fig. 3.5, or complete event-sets were corrupted. Fig. 3.6 shows an example of such an event-set.

The first idea was that the events are not in the right order, but the distribution of the timestamps over the events as a sorted plot or histogram did not show a line-like behavior. That the timestamps were right in slower motions supports the fact that the problem is not the wrong order.

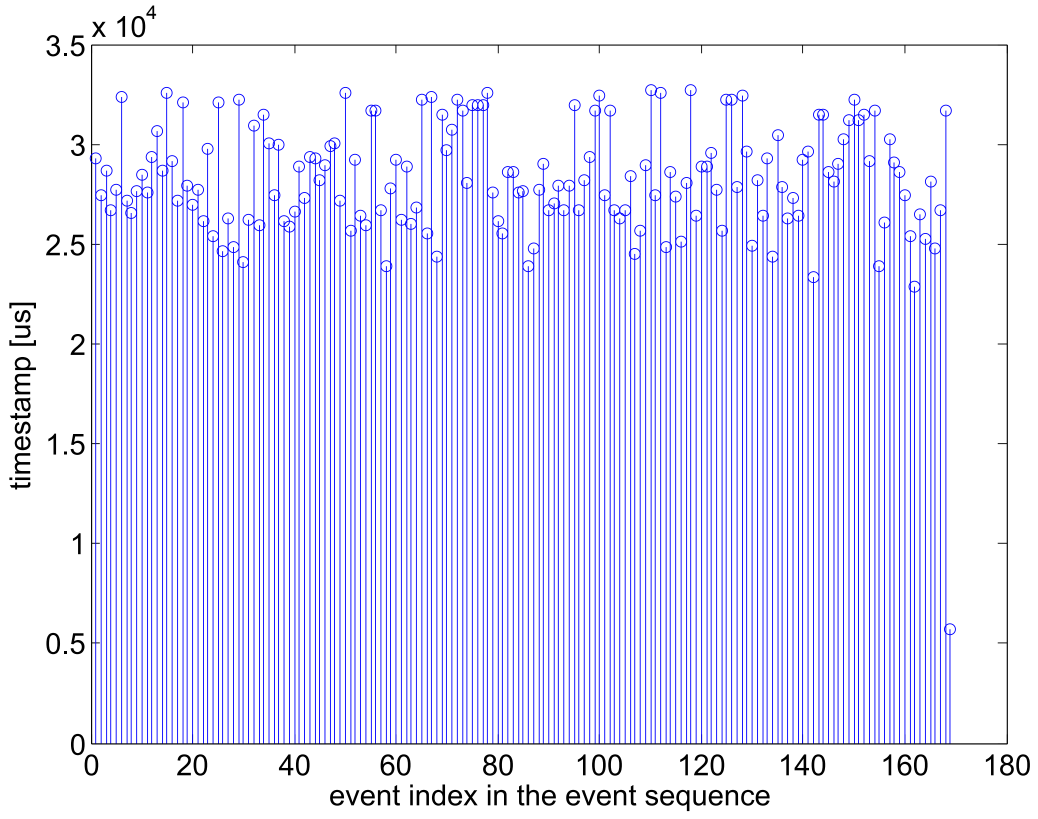


Figure 3.6: Unpredictable erroneous timestamps

The next try was to filter out timestamps that deviates too much from the previous value, from the mean or other statistical parameters. All of these approaches filtered out too many timestamps, so that the events left were not enough for the next steps or on the other side too many wrong timestamps were left and the optical flow computation was wrong.

As it happened with the USB connection, and the WLAN adapter would be even lossier and slower, it was clear then to deactivate the timestamp from the event and compute it on the PC. This has the additional advantage that the 16 Bits of the timestamp do not have to be sent from the sensor to the PC and the event transfer was noticeably faster when deactivating it. To simulate the timestamps, a linear distribution within an event-set was chosen, similar to Fig. 3.5 (except for no drop down after 65535 μs). The system time difference in μs is stopped between the arrival of the last timestamp and the actual timestamp and then divided by the number of events in this set. This leads to an equidistant linear distribution of the timestamps and works fine with the optical flow algorithm.

So all relevant data for the optical flow algorithm are available and the next step was to implement the optical flow algorithm.

3.2 Optical flow

In this section, the optical flow algorithm used for the computation of the rotational displacement is explained. The optical flow is the velocity of the object, in this case the sensor, relative to its environment [JLB05]. As it is very difficult on a quadrocopter for the optical flow to distinguish between translational and rotational motions, in this project only the rotational motions are considered [H. 09]. Fig. 3.7 shows the three different rotational motions in yaw, pitch and roll direction.

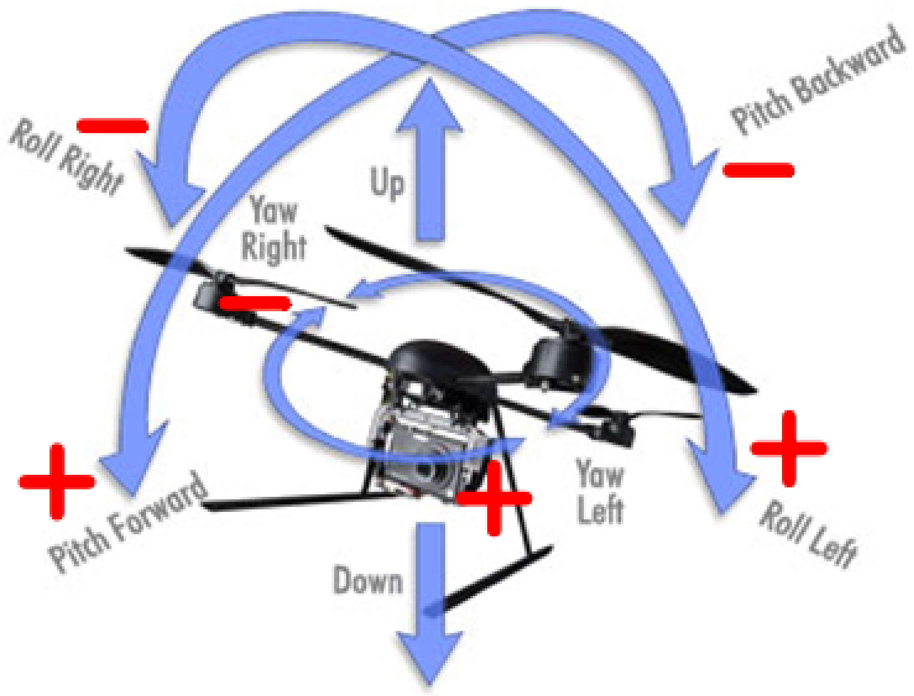


Figure 3.7: Rotational motions of a quadrocopter

As mentioned in the previous section, the algorithm here is based on sequences of events from the sensor instead of complete frames. Sequences in this case are events, which walk in time over the pixel matrix with small differences of the X- and Y-pixel index.

To understand the principle it is necessary to think about which event sequence will be get from the sensor from which motion. One sensor is placed on the top looking up. Fig. shows the optical flow directions of the sensors and Fig. 3.5 shows then the resulting sequences of events for the different motions of the drone for the sensor on the top.

[1]The real eDVS128 sensor

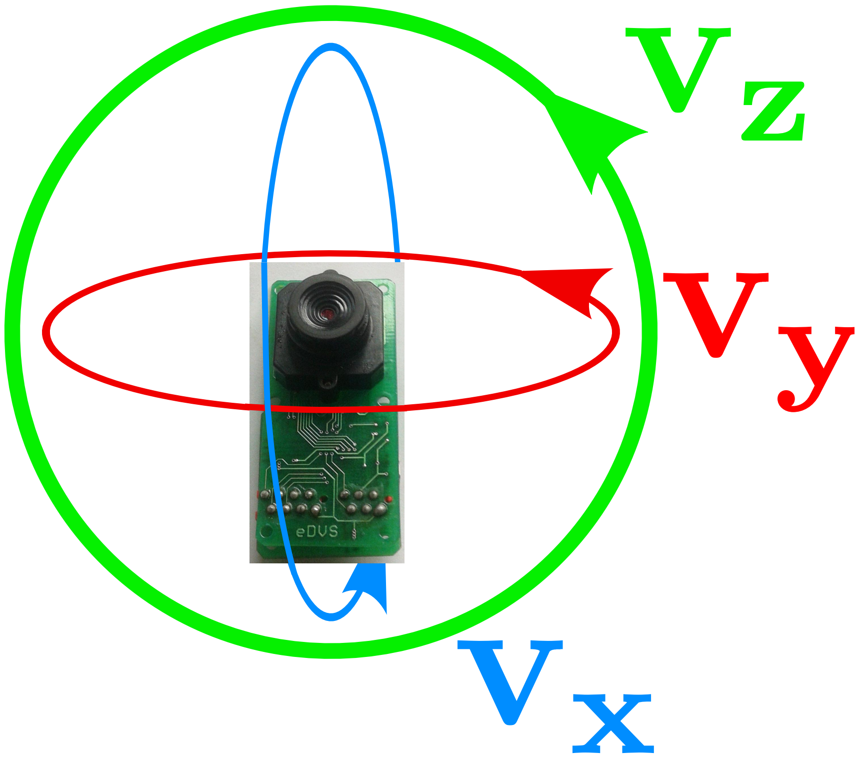


Figure 3.8: Optical flow directions for the eDVS128 sensor

For the computations here we define the origin of our starting X-Y matrix of the pixel in the lower left corner to have a common corrdinate system, where the Y-axis goes up and the X-axis to the right. As an example, the pitch motion, in this case positive when the nose of the quadrocopter goes down, will produce a sequence of events in positive direction of the Y-axis. That means there will be events along a column of the pixel-matrix with increasing Y-index and at the same time increasing timestamp. The distance between two Y-indices must be at most 3 pixels and the difference of the timestamps must be between 5 ms and 50 ms.

3.2.1 Principle algorithm

With this information a velocity in pixels per time can be computed [J. 12]:

$$v_{x,k+1} = \frac{\Delta y_{k+1}}{\Delta t_{k+1}} \quad \left[\frac{pixels}{s}\right] \quad \text{with} \quad \begin{aligned} \Delta y_{k+1} &= y_{k+1} - y_k \quad [pixels], \\ \Delta t_{k+1} &= t_{k+1} - t_k \quad [s] \end{aligned}$$

To get the rotational velocity in degrees per time, the additional information is needed how much field of view in degrees one pixel has. The whole sensor has a

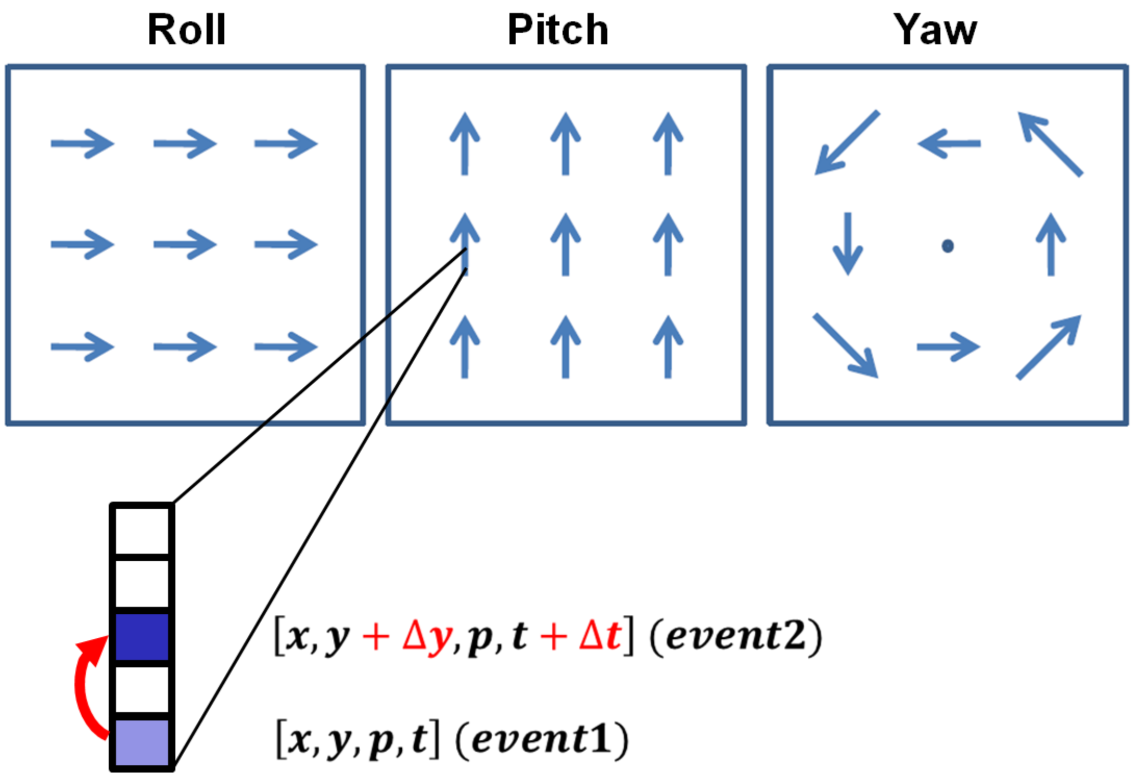

Roll
Pitch
Yaw
$[x, y + \Delta y, p, t + \Delta t]$ $(event2)$
$[x, y, p, t]$ $(event1)$

Figure 3.9: Sequences of events of roll, pitch and yaw motions from the sight of a sensor on top of the quadrocopter

range of 65° and leads to the following optical flow equation 3.1:

$$v'_{pitch,k+1} = k\ v_{x,k+1} = k\ \frac{\Delta y_{k+1}}{\Delta t_{k+1}} \qquad \left[\frac{\circ}{s}\right] \qquad \text{with} \qquad k = \frac{65^\circ}{128 pixels} \tag{3.1}$$

The roll motion is computed the same way in the X direction, whereas the yaw motion is a rotational motion in the X-Y-plane and therefore computed by:

$$\begin{aligned} v'_{pitch,k+1} &= \frac{\Delta y_{k+1} - r}{r}\ v'_{pitch,k+1} && \text{, respectively} \\ v'_{roll,k+1} &= \frac{\Delta x_{k+1} - r}{r}\ v'_{roll,k+1} && \text{with} \qquad r = \frac{128 pixels}{2} \end{aligned}$$

To smooth the signals, older velocity get into account with a decay constant d:

$$v_{pitch,k+1} = (1 - d)\ \ v'_{pitch,k+1} + d\ v_{pitch,k} \tag{3.2}$$

3.2.2 Validation

For a first validation, the displayed matrix, and optical flow velocities in the Qt GUI (Fig. 3.13) was used and the logged values during pitch, roll or yaw rotations of the sensor per hand (about 45-90°). Fig. 3.10 and 3.11 show thereby the computed optical flow. For the roll motion, the plot result looks like Fig. 3.10, only red and blue changed. Therefore the pitch and roll motions are good detected and the algorithm can be used.

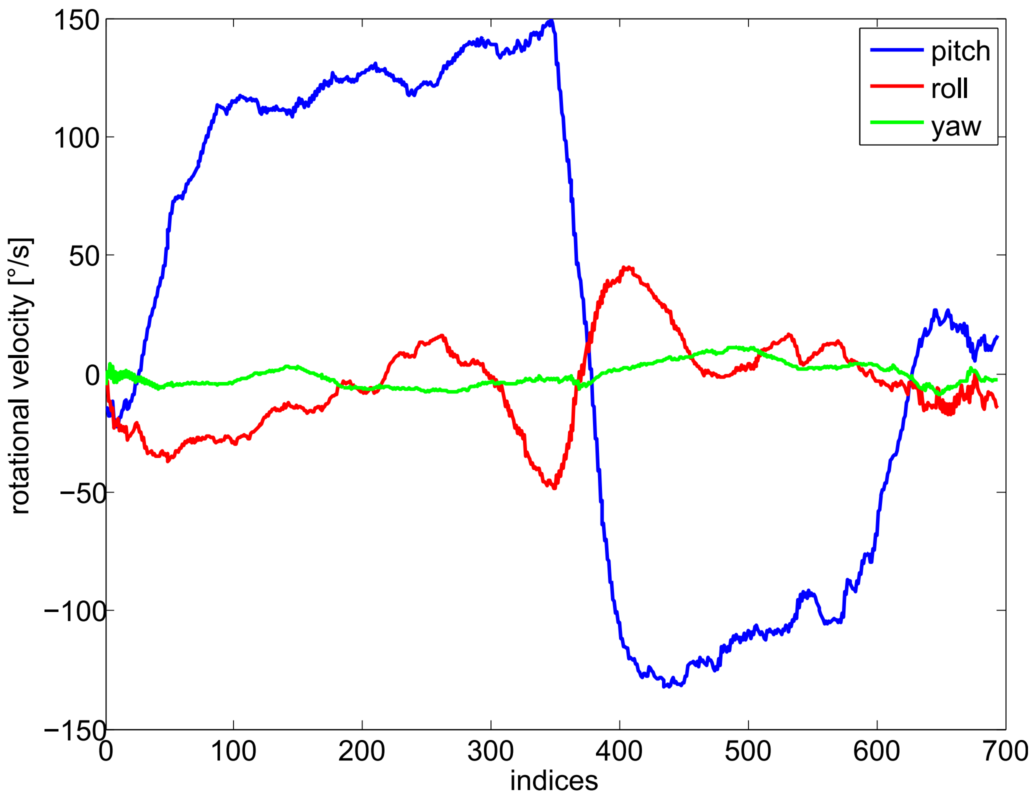


Figure 3.10: Optical flow results for a pitch motion

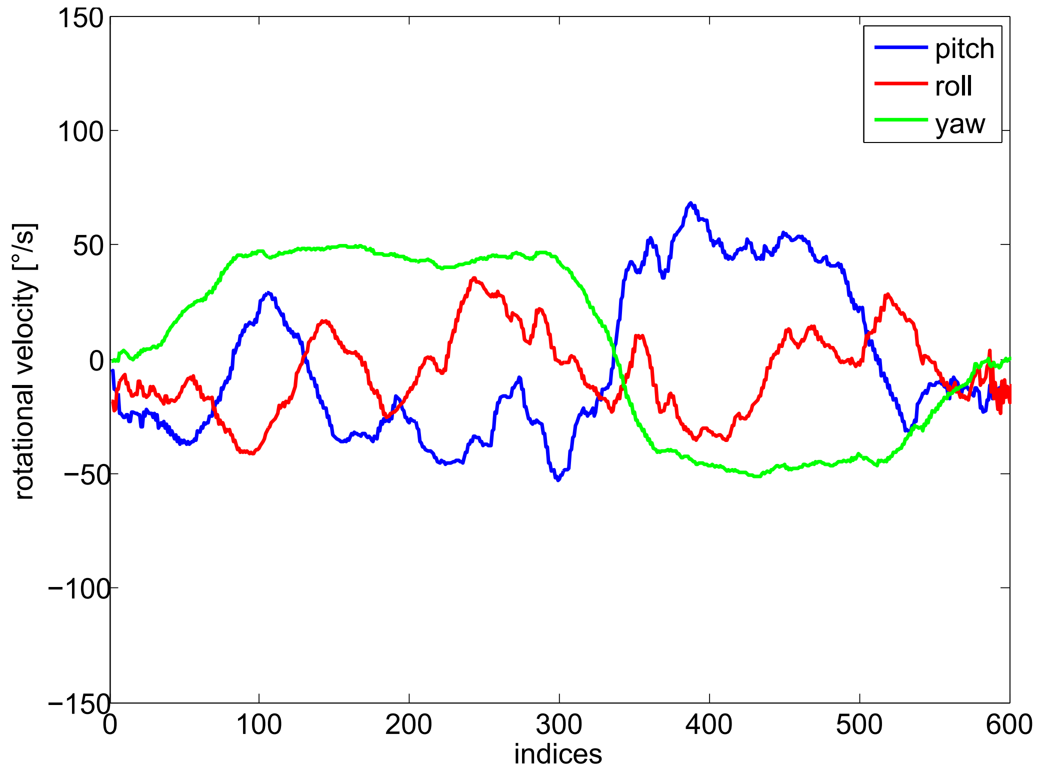


Figure 3.11: Optical flow results for a yaw motion

As it can be seen in Fig. 3.11 the computation of the yaw motion is not that reliable because pitch and roll motions are detected at the same time. It is difficult to detect because the yaw motion will always produce event sequences in the other directions in the different parts of the sensor view. But it is not that important in this project because it is concentrated on the pitch and roll motion. Therefore the second sensor is put at the opposite direction, the bottom of the quadrocopter, to also measure the pitch and roll motions at the best. This is explained together with the displacement computation in the next section.

3.3 Quadrocopter displacement

Based on the optical flow (rotational velocity in pitch, roll or yaw direction) that can be get from one sensor described in the previous section, the rotational displacement of the quadrocopter need to be computed. As mentioned before and can be seen in Fig. 3.12, the first sensor is placed on top of the drone because this project focuses on stabilizing the pitch and roll motions. These motions lead to sequences of events in the X and Y direction and can much better be recognized by the optical flow algorithm than the rotational sequence caused by a yaw motion, as explained in the previous section. In addition, a second sensor is placed on the bottom to have a redundant system leading to more reliable results in the pitch and yaw direction.

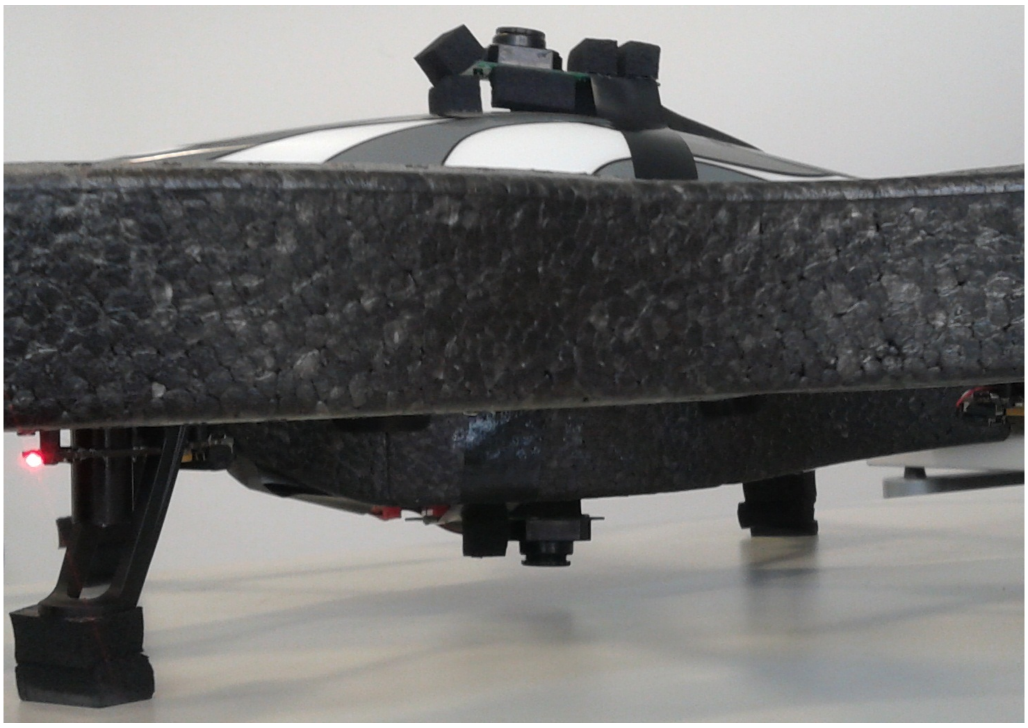


Figure 3.12: Picture of the two eDVS128 sensors attached to the quadrocopter

If the yaw motion should also be computed, another position of the second sensor would be better, for example at the side, because then the yaw motion of the drone would be recognized as events in the X or Y direction, depending on how the sensor is positioned.

To power the two sensors, they are connected parallel together to a additional battery put on the quadrocopter. As it has more than 5V and the sensors needs maximum 5V, a 5V voltage regulator is soldered in between the connecting cables.

3.3.1 Algorithm

The algorithm used for the computation is decoupled in every of the three rotational directions and the displacements can be each get from a simple integration:

$$d_{k+1} = d_k + v_{k+1,global}\ \Delta t \tag{3.3}$$

$v_{k+1,global}$ is the rotational velocity of the quadrocopter in the accordingly pitch, roll or yaw direction. To get $v_{k+1,global}$, at first the optical flow computed from the data of the sensor on the top and on the bottom of the quadrocopter is compared with the lower threshold v_{min}. If they are both below this value, $v_{k+1,global}$ in this direction is considered as 0. In this case, the displacement in this direction is not integrated but damped, to avoid the errors to sum up due to this not ideal integration. Then the displacement is computed by:

$$d_{k+1} = d_k - \frac{d_k}{decayfactor} \tag{3.4}$$

In the other case 3.3 is used. $v_{k+1,global}$ is then computed as follows. If the two computed rotational velocities of the top and bottom sensor differ too much from each other ($> \Delta v_{max}$), the values are not reliable enough and the old $v_{k,global}$ is used. Else they are averaged. Attention should be paid to the sign of the rotational velocities due to the different viewing directions of the two sensors. Therefore $v_{k+1,global}$ is computed like this in the different directions:

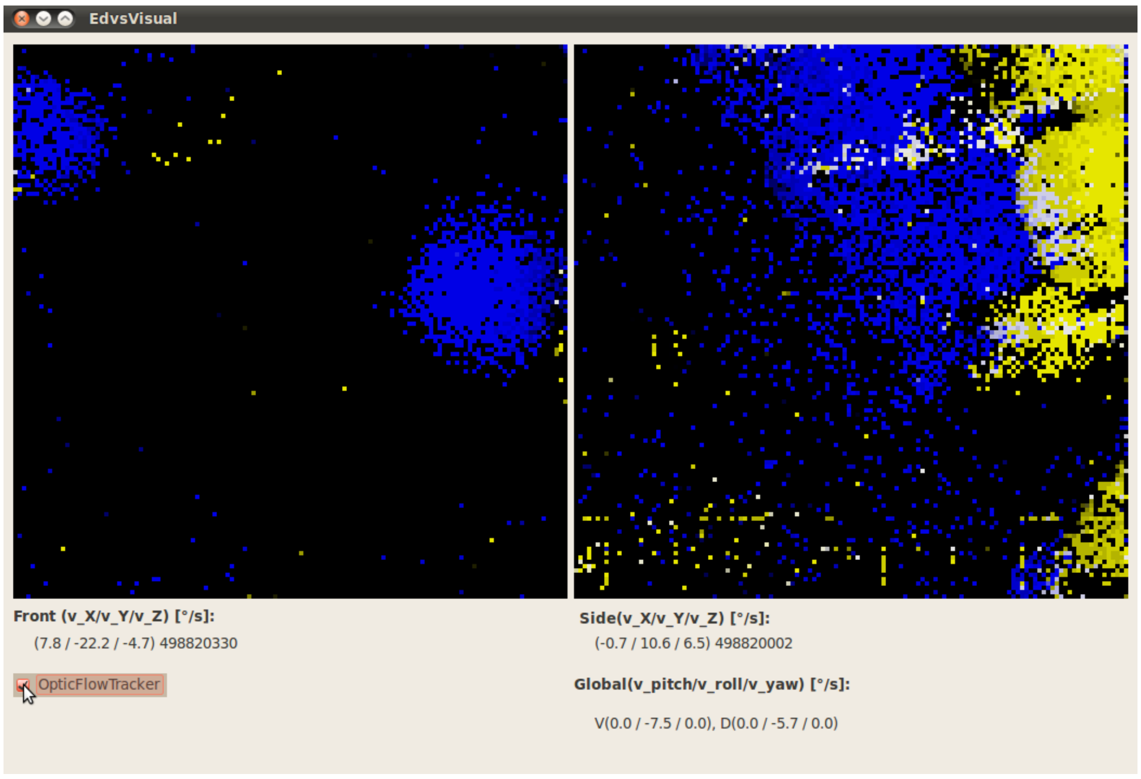
$$v_{pitch,k+1,global} = \begin{cases} v_{pitch,k+1,global} & \text{if } v_{pitch,k+1,top} - v_{pitch,k+1,bottom} > \Delta v_{max} \\ \frac{v_{pitch,k+1,top}+v_{pitch,k+1,bottom}}{2} & \text{else} \end{cases}$$

$$v_{roll,k+1,global} = \begin{cases} v_{roll,k+1,global} & \text{if } v_{roll,k+1,top} + v_{roll,k+1,bottom} > \Delta v_{max} \\ \frac{v_{roll,k+1,top}-v_{roll,k+1,bottom}}{2} & \text{else} \end{cases}$$

$$v_{yaw,k+1,global} = \begin{cases} v_{yaw,k+1,global} & \text{if } v_{yaw,k+1,top} + v_{yaw,k+1,bottom} > \Delta v_{max} \\ -\frac{v_{yaw,k+1,top}-v_{yaw,k+1,bottom}}{2} & \text{else} \end{cases}$$

3.3.2 Validation

Particularly for the parameter adjustment and algorithm validation, the displayed values of the separate optical flow velocities and computed roll-, pitch- and yaw velocities and displacements in the Qt GUI were used when moving the quadrocopter (Fig. 3.13).
Most of the validation was done by moving the quadrocopter per hand in the different directions. This is because the quadrocopter cannot fly a greater specific rotational motion without translating too much and therefore distorting this results. Only after



Front (v_X/v_Y/v_Z) [°/s]:
(7.8 / -22.2 / -4.7) 498820330

Side(v_X/v_Y/v_Z) [°/s]:
(-0.7 / 10.6 / 6.5) 498820002

Global(v_pitch/v_roll/v_yaw) [°/s]:
V(0.0 / -7.5 / 0.0), D(0.0 / -5.7 / 0.0)

Figure 3.13: Screenshot of the running Qt GUI

having the parameters roughly adjusted, real flights were logged. Here only pitch motions are considered, because as descripted in the optical flow section, the results of the roll motion behave similar to the pitch motion, and yaw motions are not considered in this project. But what can be said about the parameters for the yaw motion is that because of the smaller recognition of this optical flow (see FIG.), the thresholds need to be more sensitive (v_{min} and Δv_{max} smaller).

At first the best range for the thresholds Δv_{max} and v_{min} for the velocity computation were tested because this is not direct dependent with the displacement (only reversed). Δv_{max} seems not to have such a big influence when not adjusted that exactly, although it was better to be higher at about less or equal to 100 at a start. By no means should it be too small. In this case the existing difference between the optical flow from the top and bottom sensor would lead to way too many filtered out values.

Much more influence has the threshold v_{min}. That it is much more sensitive can be seen in Fig. 3.14 and Fig. 3.15. In Fig. 3.14 with $v_{min} = 10$ there is a lot of noise, whereas in Fig. 3.15 with $v_{min} = 25$ the noise is nearly filtered out completely. However this is done by the cost of cutting the zero transition, which means small velocities, of the real pitch motion (blue line). If this could lead to problems for the controlling at the end should be tested, but at the end the value should be, if any, only a bit under $v_{min} = 25$ to have not too much noise.

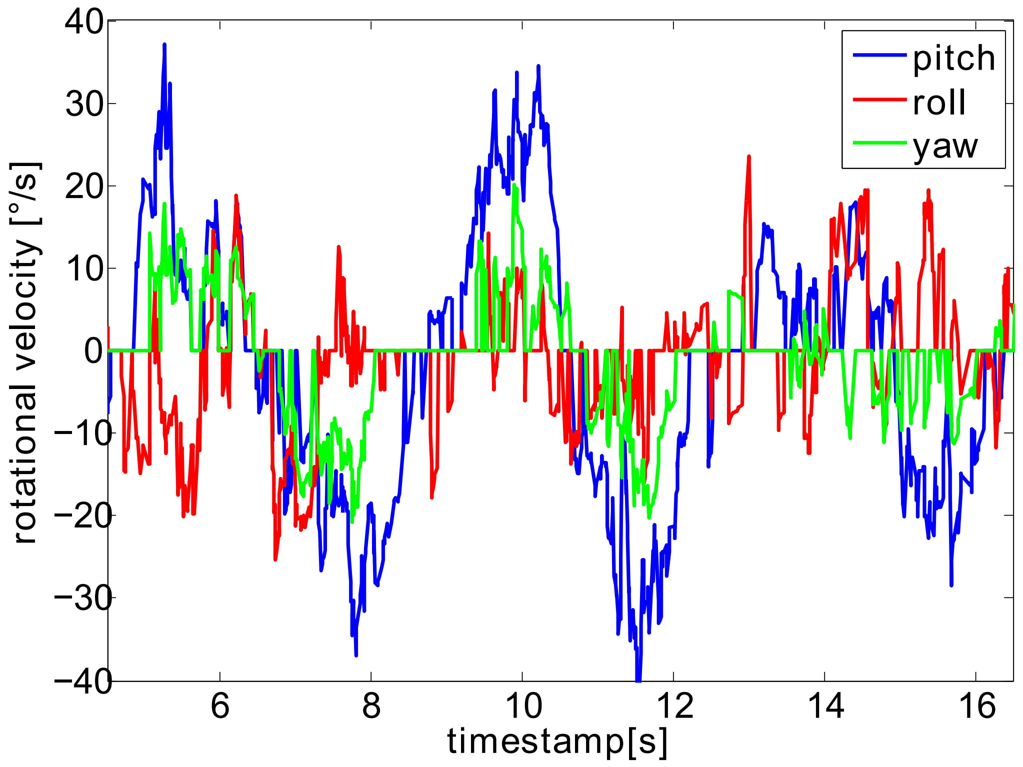


Figure 3.14: Computed rotational velocities over time with $v_{min} = 10$ and $\Delta v_{max} = 100$ during pitch rotations of the quadrocopter moved by hand

The recognized motions in the roll and yaw direction in Fig. 3.15 are because of the not possible ideal pitch motion per hands without turning also a bit in other directions.

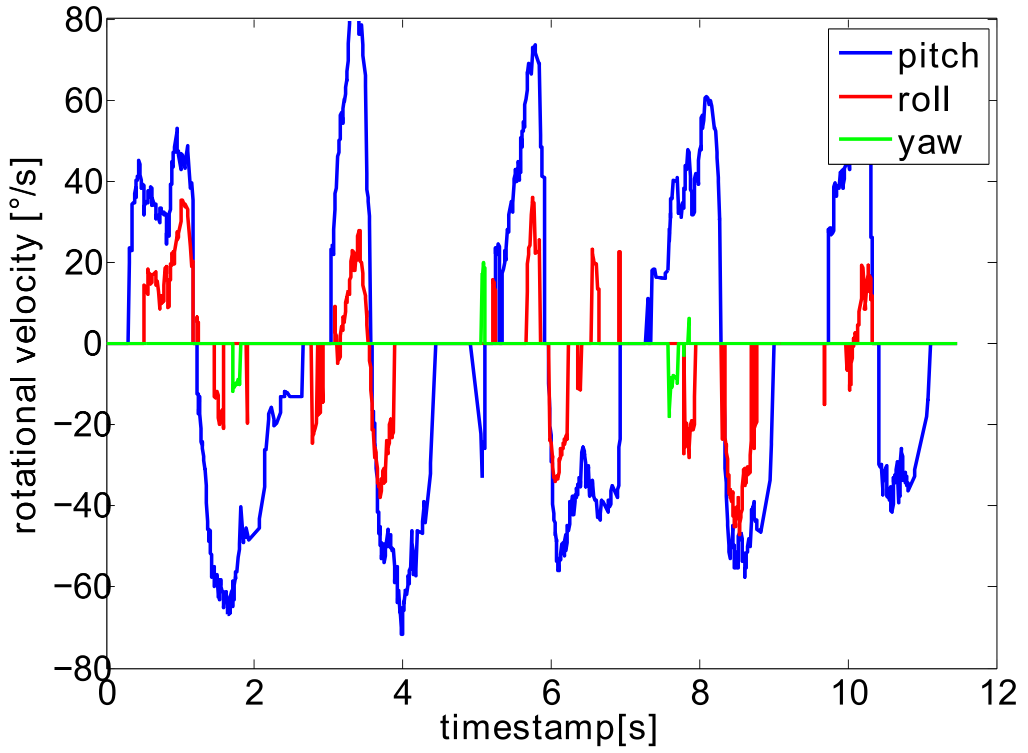


Figure 3.15: Computed rotational velocities over time with $v_{min} = 25$ and $\Delta v_{max} = 100$ during pitch rotations of the quadrocopter moved by hand

After testing the parameters for the velocity computation, the parameter $decayfactor$ for the displacement computation has to be tested. As it can be seen in Fig. , no decay or in this case a very small decay (high $decayfactor$) leads to a residual error. Because the pitch motion was a turning of about 45-90°and back to the initial orientation, the displacement should also go only back to 0 and not below zero. In Fig. , this is constantly more and more suspending below zero. In addition, the computed displacement has a huge delay to the real motion, as it can be seen when comparing with the time when the velocities have the according values.
However if the $decayfactor$ is extremly small (high decay) (4 in Fig.), the displacement indeed goes always immediately back to zero, but at the same time the amplitude is damped. Even worse is that it is so fast decaying that it goes even to zero when the rotation is at its highest point and the velocity therefore 0. The consequence is that the back movement to the initial position is interpreted as a movement in the other direction. Even if the $decayfactor$ is 100 as in Fig. , where the damping of the amplitude is gone, the other problem is still there.

For this reason, it should be tested as a next step with the real quadrocopter and controller which value between 10000 and 100 has for this application the best compromise between summing up error and too much and fast decay.

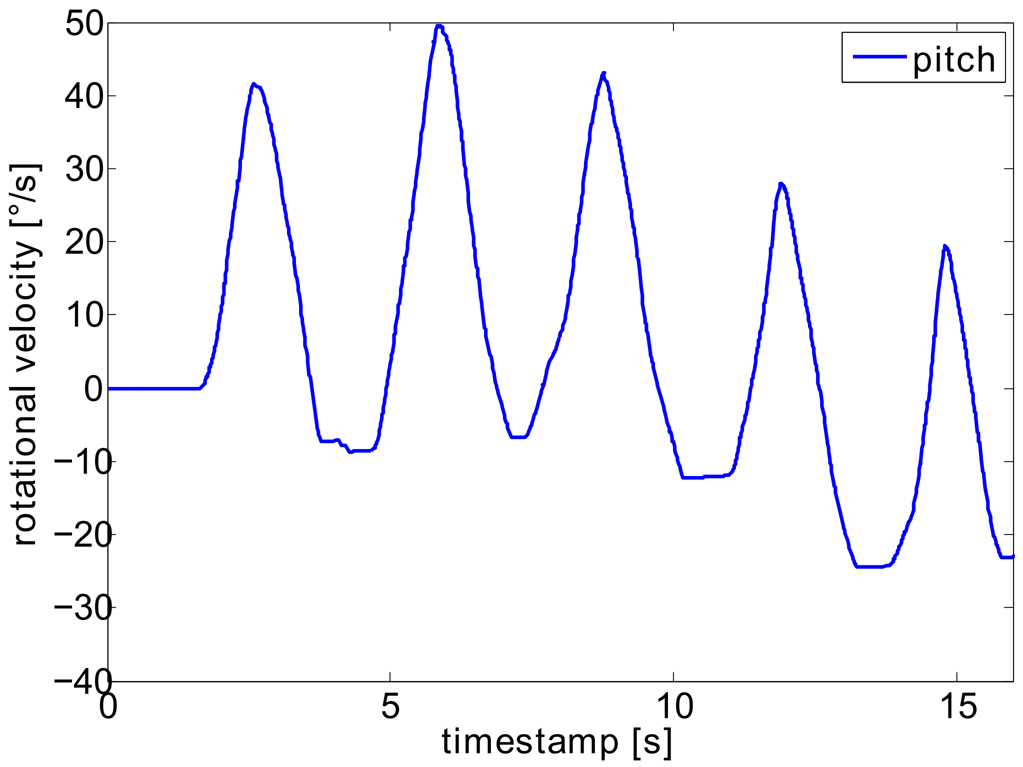


Figure 3.16: Computed rotational velocities over time with $decayfactor = 10000$ during pitch rotations of the quadrocopter moved by hand

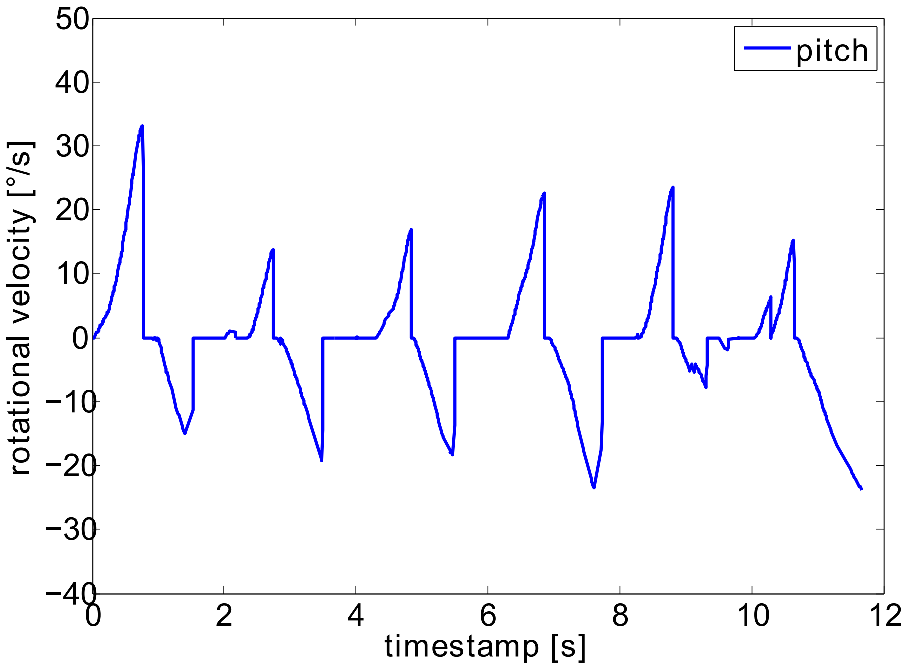


Figure 3.17: Computed rotational velocities over time with $decayfactor = 4$ during pitch rotations of the quadrocopter moved by hand

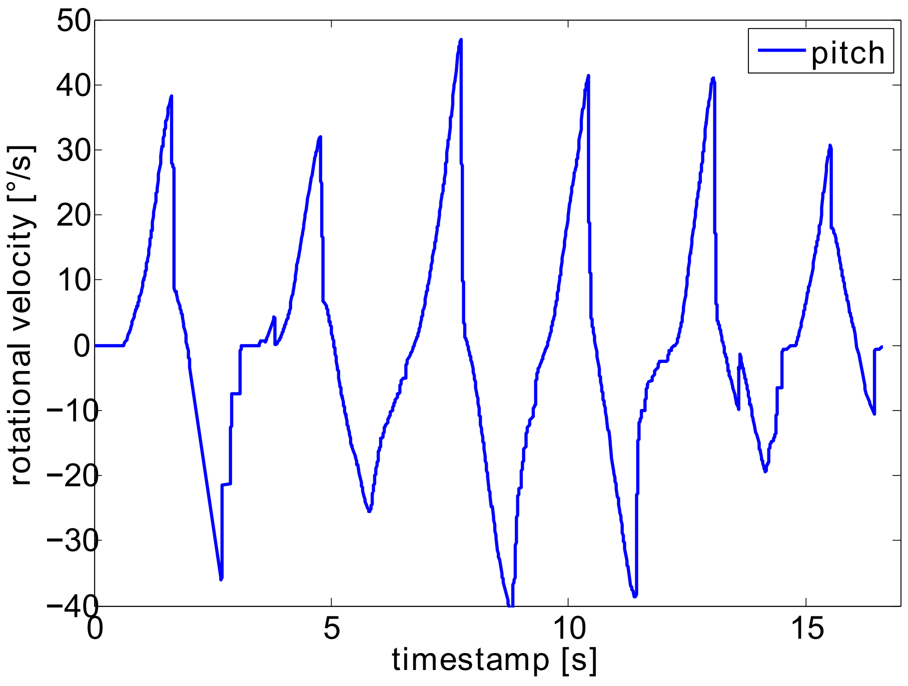


Figure 3.18: Computed rotational velocities over time with $decayfactor = 100$ during pitch rotations of the quadrocopter moved by hand

Kapitel 4

Summary (Franz)

In this project we could develop a stabilization of a quadrocopter with assistance of an embedded neuromorphic vision sensor. We wrote a control programm, with which we are able to test the stabilization, by applying massive disturbances on the drone. We use the event based data from the eDVS sensor, to compute with an optical flow algorithm the deviation of the roll and pitch angle and the angular velocity.
With this work we could demonstrate the wide appliation field of the eDVS sensors. Future work could try to create a on-board stabilization without use of any other sensor, or to control translational movements.

Abbildungsverzeichnis

Literaturverzeichnis

[H. 09] H. Romero, S. Salazar, R. Lozano, editor. *Real-Time Stabilization of an Eight-Rotor UAV, Using Optical Flow*, 2009. IEEE TRANSACTIONS ON ROBOTICS, VOL. 25, NO. 4, pp 809-817.

[J. 12] J. Conradt, editor. *Optic Flow from Miniature Event-Based Vision Sensors*, 2012. Technische Universitï¿½t Mï¿½nchen.

[JLB05] N. A. Thacker J. L. Barron. *Tutorial: Computing 2D and 3D Optical Flow*, 2005. University of Manchester.

[M. 07] M. Oster, P. Lichtsteiner, T. Delbruck, Shih-Chii Liu, editor. *A Spike-Based Saccadic Recognition System*, 2007.

[NTa] NST-TUM. Getting started with edvs128. `https://wiki.lsr.ei.tum.de/nst/programming/edvsgettingstarted`.

[NTb] NST-TUM. Wlan module redpine rs9110-n-11-22. `https://wiki.lsr.ei.tum.de/nst/setup-redpine/index`.

[P. 07] P. Lichtsteiner, C. Posch, T. Delbruck, editor. *An 128x128 120dB 15us-latency temporal contrast vision sensor*, 2007. IEEE J. Solid State Circuits, 43(2), 566-576.

[Para] Parrot. Ardrone api. `https://projects.ardrone.org/projects/show/ardrone-api`.

[Parb] Parrot. Ardrone technology. `http://ardrone.parrot.com/parrot-ar-drone/de/technologies`.

[ZR11] S. Zacher and M. Reuter. *Regelungstechnik fuer Ingenieure.* 2011.