DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Development of a drone-based evaluation tool for motion analysis in athletics long jump

Studienarbeit

im Studiengang

Informationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Name, Vorname:	Faust, Jakob
Abgabedatum:	16.04.2024

Bearbeitungszeitraum:	17.10.2023 - 16.04.2024
Matrikelnummer, Kurs:	5507125, TINF21IT1
Betreuer:	Jürgen Schultheis

Unterschrift Betreuer: _____

Mannheim, den

Contents

List of Figures

Listings

List of acronyms

AI Artificial Intelligence

BEC Battery Elimination Circuit

CM Center of mass

CNN Convolutional Neuronal Network

CPU Central Processing Unit

ESC Electronic Speed Control

FPS Frames Per Second

FPV First Person View

GPS Global Positioning System

GPU Graphical Processing Unit

GUI Graphical User Interface

HAT Hardware Attached on Top

HPE Human Pose Estimation

PDB Power Delivery board

PM Power Module

PWM Pulse Width Modulation

RPM Revolutions Per Minute

SSE Sum of Squared Erros

TCP Transmission Control Protocol

UDP User Datagram Protocol

1. Introduction

Long jump is an athletic discipline that is renowned for its technical complexity and the precise movement patterns it demands from athletes. Even apparently small technical inaccuracies can significantly impact an athlete's performance. Moreover, as shown in [1] the forces during the take-off phase can reach up to 10 times the athlete's body weight, increasing the risk of serious injuries due to technical inaccuracies. Therefore, it is crucial to understand and continuously improve these movement patterns in training. However, taken the high approach velocity[1] into account, this can quickly become a difficult task. Especially the take-off phase can be very short and therefore hard to analyze.

Professional athletes often employ expensive high speed camera systems in combination with body pose markers to capture and analyze every single step they make.
Yet, this approach comes with some limitations. Due to their stationary installation, such camera systems are restricted to a fixed location. Moreover, they often combine multiple cameras like Murray et al. [2] used for sprint analysis in order to be able to capture the whole movement from the beginning of the approach until the landing. This leads to complex post-processing software requirements. Additionally, fixed markers need to be attached to an athletes body to be able to track their body position.

While these methods provide exact and reliable results, they are usually not accessible for hobby- and semi-professional athletes.
In recent years however the advances in Artificial Intelligence (AI) and especially within the area of deep neural network paved the way for analyzing methods that require less complex setups. As of 2023 deep neural networks trained for body pose detection are even used in medical applications like gait analysis [3]. Because of the already extremely high and continuously improving accuracy, its application within the area of motion analysis in long jump is treated in the scope of this work.
A semi-autonomous drone based evaluation tool is newly developed. It is supposed to offer a portable alternative to address the lack of existing opportunities in analyzing long jump performances in training. For this purpose, the drone should autonomously fly next

[1] around 10 m/s in male semi-professional long jump

to the athlete throughout the whole jump, capturing their motion and therefore allow for a complete jump analysis. The drone itself is based on First Person View (FPV) drone hardware. It is build from scratch using an onboard single-board computer as flight control unit responsible for capturing the video. Additionally, a ground station software is developed to allow for a convenient jump analysis regarding the overall body pose as well as a fixed set of important parameters, i.e. knee angles, arm angles, hip position. The project's source code is available under `https://github.com/JF631/FLYJUMP`.

2. Tasks

This chapter provides an overview over all tasks that are approached in this project. Hardware related tasks are listed as well as software related tasks. Each main task is divided into several sub-tasks.

Task 1 Develop a platform independent computer software to analyze pre-recorded long-jump videos.

1. Implement a video processing pipeline to analyse a jump regarding a fixed set of parameters.
2. Develop an algorithm to automatically detect the take-off frame.
3. Define and implement an analysis output standard.
4. Validate the performance to allow on-field jump analysis
5. Implement a GUI to visualize the analysis results.

Task 2 Build a quadcopter as hardware companion for recording long-jump videos. This specifically requires a camera as well as several sensors for navigation.

1. Select hardware components according defined criteria.
2. Build a flying quadcopter that is controllable from a ground station.
3. extend the quadcopter with a companion computer and a compatible camera to live stream videos.

Task 3 Implement a software that allows to control the drone from a ground station.

1. Set up a wireless connection to the quadcopter.
2. Implement a drone control offering movement control.
3. Integrate the drone control in the GUI

Task 4 Develop a software to handle the incoming video livestream.

1. Establish a connection to the on-board companion computer.
2. Display the video frames in the GUI.
3. Offer the option for a rudimentary live analysis.

3. Methodology

The following chapter provides an overview over the relevant development components that are used within this project. Therefore, the used software packages are introduced before a short outline of the utilized drone hardware is given.

3.1. Software fundamentals

As the main part of this project's software will run on a portable remote computer allowing for not only to control the drone but also for performing the long jump analysis on video inputs, every software component is chosen to demand as little hardware requirements as possible. Especially no Graphical Processing Unit (GPU) is required to run the software. All image processing is performed using the Central Processing Unit (CPU) only. Furthermore, the software is designed to run platform independent.

3.1.1. Programming Language and why Python

Because of its interpreted nature and many cross-platform libraries, Python is one of the most used programming languages in the scientific area. Furthermore, it offers a high level of abstraction allowing for rapid prototyping approaches which is a key factor for this project. Besides fast implementation, Python nevertheless supports complex programming concepts like object-orientation.
Additionally, as stated in subsection 3.1.2 many machine learning and AI projects for detecting body poses have already been successfully implemented using Python.
Third party libraries and frameworks like *OpenCV* for image processing and *PyQt* for Graphical User Interface (GUI) development are widely used and therefore well documented. This leads to the decision to use Python as programming language within this work.

3.1.2. Mediapipe for detecting body poses

One of the software's main tasks is to perform a human body pose detection on video inputs. Because this part runs on the remote computer only, it can also handle pre-recorded

videos that should be evaluated.
The evaluation itself is performed using the Mediapipe framework. This framework uses a pre-trained convolutional neural network that is able to detect 33 key points in human body poses [4]. The network could theoretically even be fine-tuned to improve its accuracy on specific input types. Even if this so called *transfer-training* method requires significantly less training data compared to training a neural network from scratch, it is not applied within this project as first test runs already showed reliable results.
Furthermore, the Mediapipe framework does not require any hardware acceleration and is renowned for its precise output. Hii et al. for example showed in [3], [5] that the framework can be applied in medical gait analysis applications to replace marker based approaches. Moreover, Mediapipe offers three different detection models that differ in terms of speed and accuracy. The fastest detection model offers the lowest accuracy and vice versa.
Additionally, Mediapipe is optimized for multiple input types including videos and live streams, which is ideal for this project.
Figure 3.1 shows an overview of the 33 detectable key points.

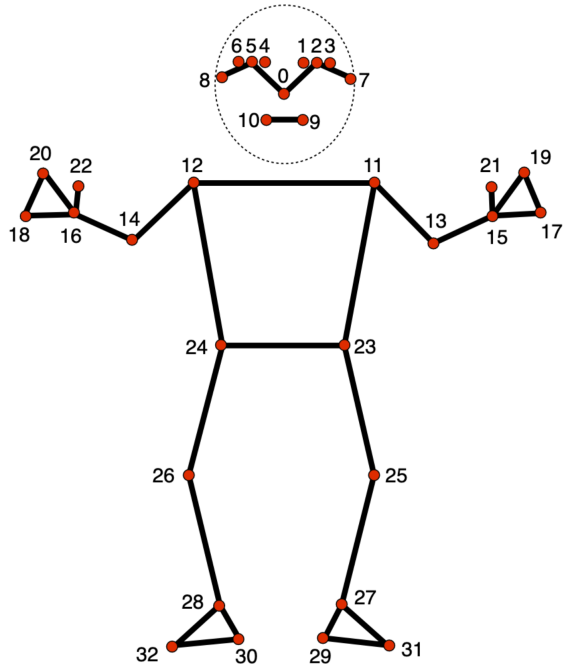


Figure 3.1.: Fixed set of detectable body key points offered by the mediapipe framework [6]

Within this work, the key points in the head area (range [0 - 10]) are not of great interest apart from visualization purposes.
The knee, hip and arm key points however will be used for angle calculations and ground contact detection. Thus, a good performance in detecting the according key points within these areas is crucial for the software's overall reliability.

3.1.3. Why Mediapipe?

In recent years many approaches towards accurate body pose detection were developed and implemented. Many of those offer decent accuracies, but often lack reasonable performance, especially when no GPU is available for hardware acceleration. Following, two common alternatives to Mediapipe, namely OpenPose and AlphaPose, are shortly presented and differentiated from the chosen Mediapipe framework.

One of the most widely used human pose detection libraries is the open source library *OpenPose*. As shown in [7] it offers a Multi-Person pose estimation that is especially useful when dealing with groups of people. However, as this project is meant to be used for long jump evaluation, only one person needs to be tracked at a time. Even though OpenPose of course can handle one person pose estimation, mediapipe outperforms OpenPose in this area. Back in 2016 Kocabas et al. achieved around 23 FPS using GPU accelerated OpenPose pose detection [8] and Osokin later proposed an improved neural network design, allowing for up to 28 FPS without hardware acceleration [9]. As of 2023 these benchmarks are still reasonable. Mediapipe however achieves speeds of up to 63% higher.

While OpenPose uses a *Bottom-Up* approach due its multi-person application, Mediapipe uses the less computational complex *Top-Down* approach.

Bottom-Up implementations first detect all body key points present in an input image and then move on to grouping the recognized points in clusters. Points in the same cluster are then assigned to one person.

Top-Down approaches however first roughly detect the overall body position within the input image and then define a region of interest[1] around the subject. The following processing therefore only needs to take this defined region of interest into account, leading to significantly less computational complexity.

AlphaPose is another open source library often used for body pose estimation. Just like OpenPose it uses a Bottom-Up approach to reliable offer multi-person body pose detection. Additionally, AlphaPose, like Mediapipe, offers multiple detection models that differ regarding accuracy and speed. Again however, Mediapipe outperforms AlphaPose because of its Top-Down approach and because AlphaPose is designed to work with GPU acceleration, rather than running on CPU only.

Another advantage of Mediapipe is its output. While OpenPose and AlphaPose offer 2D coordinates for each detected key point, Mediapipe additionally offers a depth estimation resulting in a spatial 3D coordinate for each detected key point. Thereby, more comprehensive analysis can be performed.

[1] sometimes also referred to as *Bounding Box*

3.1.4. OpenCV

OpenCV is an open source library commonly used for image processing in the area of computer vision and machine learning. It is written in C++, thus offering high performance in numerical operations, especially matrix operations.
`python-opencv` is the python wrapper for OpenCV which will be used in this project to efficiently read and process video frames. The python wrapper imports the underlying C++ functions as modules to take advantage of C++'s efficiency, resulting in significantly higher performance compared to equivalent Python only implementations. Furthermore, it is fully compatible with the `numpy` library, which allows for seamless conversion between numpy arrays and OpenCV image matrices.

3.1.5. HDF5 file format

After a jump analysis is completed, there are two files to be stored. One video file containing the actual body key point video analysis by showing the key points as overlay and another one holding the analyzed parameters. The second file especially requires the file format to be structured in order to be able to allocate the stored analysis parameters to their according video frame.
One common open source structured file format is HDF5[10]. It is an acronym for Hierarchical Data Format (Version 5) and is very efficient to store large amounts of data as well as heterogeneous data. As the name already suggests, data is stored in a hierarchical, tree-like way.
A HDF5 tree mainly consists of three pre-defined components that allow to organize data in a file-system like fashion. Namely, those components are the tree root, groups and datasets. *Groups* are folder-like structures that can either contain more groups or Datasets. *Datasets* hold the actual data that need to be stored.
Furthermore, each level (tree root, groups and datasets) can hold additional information via metadata. The metadata could, for example, contain information about metrics, timestamps or any other describing information. Therefore, each HDF5 file itself becomes a self-describing file that does not require any more than the included information to be interpreted correctly.
Thus, it is chosen as primary analysis parameter storage format within this project.

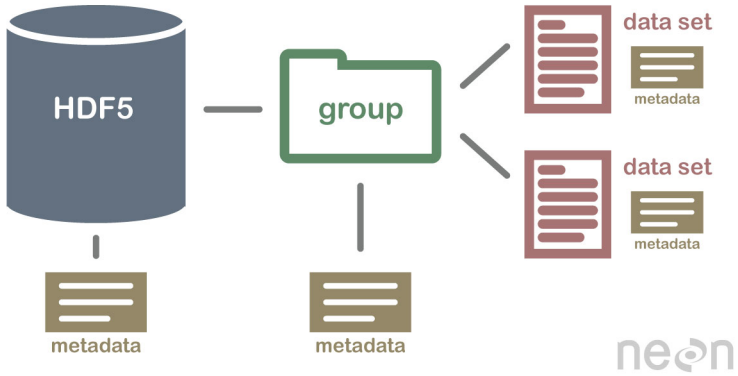


Figure 3.2.: Principle HDF5 file structure [2]

Within this work, one HDF5 file will be created per jump analysis.

3.1.6. Precompiling Python code using Numba

Built-in Python functions as well as some `numpy` functions can be sped up using the Python *Just-in-time compiler* `numba` [11]. This compiler is based on the LLVM compiler structure and generates CPU specific, optimized machine code. This is especially helpful in numerical heavy parts of the software, which can significantly benefit from pre-compiled code regarding their runtime behavior.

3.1.7. Software overview

The software within this work is implemented using python (see subsection 3.1.1). As it supports a wide range of software features, a variety of different packages is used. The most relevant packages and frameworks are numpy for efficient numerical operations, h5py for convenient HDF5 file operations and PyQt5, which is used for the GUI development. Moreover, the software has been implemented and tested using Python 3.11.5 and is backward compatible up to Python 3.9. Python 3.12 is explicitly not supported, as the mediapipe framework not yet supports this version. As the packages are combined into one software, an overview of the used packages is given:

Software package overview	
Package	Version number
Jump analysis	
mediapipe	0.10.7
numpy	1.26.1
numba	0.59.0
scipy	1.12.0
opencv-python	4.8.1.78
psutil	5.9.6
h5py	3.10.0
GUI development	
PyQt5	5.15.10
PythonQwt	0.11.1
Drone control	
pymavlink	2.4.41
pyserial	3.5

Table 3.1.: Software package overview and their according version numbers.

3.2. Related work

As more advanced machine learning models are available, Human Pose Estimation (HPE) is a quickly growing field in the area of sports video analysis. There are generally two common approaches which have been established over the past few years. Namely, these are the more recently proposed Transformer [12] based HPE models and the longer established Convolutional Neuronal Networks (CNNs).
Mediapipe[4] belongs to the CNN based models. Leddy et al. [13] successfully used this framework to calculate joint angles in sport videos. Similar approaches using CNN based architectures have been used by Yu et al. [14] to analyze standing long jumps. Zhou et al. [15] followed a similar approach to analyze athletes' motions during a long jump, starting from the takeoff. However, the mediapipe pose detection model has also recently been applied in other sport related fields. Aarthy et al. [16] used it to identify and analyze yoga positions which is more of a static task compared to long jump videos.
Transformer based approaches are a more recent development which, instead of taking (small) image regions into consideration for feature extraction, take all pixels of an image into account to weigh their importance relatively to each other. Generally, these approaches are restricted in the set of key points they can detect. Recent developments [17], [18] however paved the way for more complex body joint detection tasks. Building up on this, Ludwig et al. [19] developed a network capable of detecting arbitrary body key points in jumping videos.

3.3. GUI development

The software that is developed within this work should be useable on-field to allow for a fast analysis process. Besides the video analysis, the drone control should be embedded in the same software to always guarantee control over the drone.
Thus, a simple GUI is developed to offer a convenient drone control and video analysis process.
The GUI is developed using Qt and its Python binding PyQt. Both are presented in this section.

3.3.1. Development framework Qt

Qt is a development framework based on the programming language C++. It includes a GUI toolkit and therefore enables platform-independent application development. All common platforms including Linux, Windows and MacOS are supported by Qt. Additionally, both mobile operating systems, Android and iOS, are supported.
This project however will focus on the development of an application that is able to run under Windows, Linux and MacOS.
The Qt framework is mainly chosen because of its rich and comprehensive documentation[3] and the availability of the well-supported Python binding *PyQt*.
As Qt is based on C++ all Qt source files are translated to C++ code. This step is realized by the `Meta Obejet Compiler (MOC)` which is integrated as pre-processor. Thus, all Qt files are translated to so-called *Meta Obejet Code*, which can be seen as C++ source code with some enhancements. The most important enhancement is the signal and slot functionality which allows for an easy communication between different software and design elements (e.g. show a message dialog when a button is clicked).
Another important enhancement for this work is the convenient multi-threading management necessary for offering a responsive GUI even when cpu-bound calculations such as image processing is performed.

The GUI module PyQt

The discrepancy between Qt as C++ based framework and Python as chosen programming language for this project (as explained in subsection 3.1.1) can be overcome using Qt's Python binding PyQt. By using PyQt we can combine Python's machine learning advantages with Qt's platform-independent GUI development. More specifically *PyQt5* [4] is used.

[3] `https://doc.qt.io/`
[4] `https://pypi.org/project/PyQt5/`

It allows building complex Qt applications using Python only instead of C++. All other described advantages that Qt offer remain valid despite the use of PyQt. Thus, the whole software within this project including the GUI can be developed using Python as programming language.

4. Implementation

This chapter focuses on the overall project's implementation. It mainly covers the four parts hardware assembly, long jump analysis, drone control and their consolidation into one GUI.

4.1. Long-jump analysis

In order to analyze recorded long jump videos a ground station software is developed. Generally, the analysis is performed regarding the following set of parameters:

- left / right knee angle
- left / right arm angle
- takeoff angle
- left / right foot position
- hip height

These parameters are tracked over the whole jump, beginning with the run-up throughout the takeoff until the landing.
Figure 4.1 shows a detailed overview over the video data that can be analyzed by the software.
As the takeoff is (one of) the most important phases in a long jump, it is important to be able to detect the takeoff in a video. Such a takeoff detection is developed alongside the above-mentioned parameter detection- and calculation.
This section will first introduce the body key point detection process that is basis for all following calculations. Afterwards, an algorithm for an automatic takeoff frame detection based on a video input is implemented.

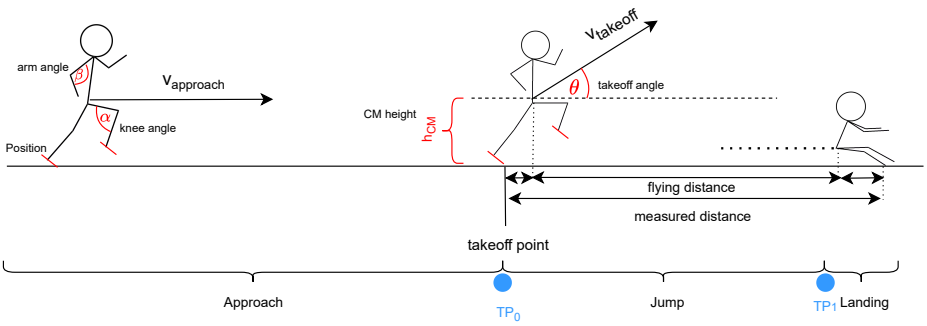


Figure 4.1.: General long jump overview.
Parameters that can be analyzed are marked red.
TP_0 and TP_1 denote phase transition points.

4.1.1. Video processing

After a video has been recorded, several processing steps need to be performed to calculate the parameters shown in Figure 4.1. All calculations are performed for each input video frame respectively. In the first processing stage, multiple filters can be applied to each frame. The body key point detection is then performed on the filtered output. Based on these key points, the jumping parameters can be calculated which are then in a last stage saved to a hdf5 file as well as used for the automatic takeoff frame detection.
The whole process is visualized below in Figure 4.2.

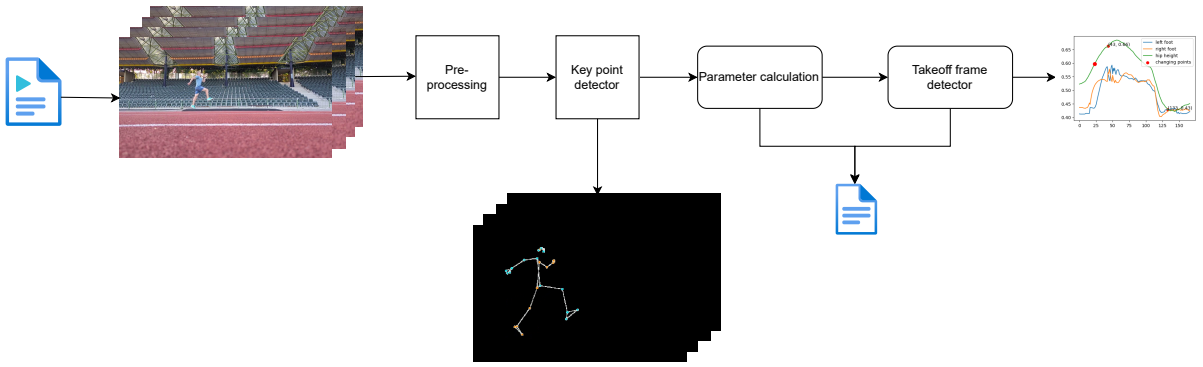


Figure 4.2.: Video processing pipeline.
Stages, that each video frame passes through.

In the following subsections each of the shown processing stages will be explained in detail.

Pre-processing via filter application

The first stage is meant to prepare the incoming video frames to get the best possible result from the following pose detection stage. Thus, this stage will in the following be referred to as *pre processing stage*.

To get an accurate pose detection result it is important to have sharp edges around the athletes' body and an overall low noise level. This can be achieved by applying different filters to each frame.
Three of the most commonly used filters are high-pass (sharpening), low-pass (blurring) and edge preserving low-pass filters, also known as bilateral filters. As the best choice depends on the video input and recording conditions, all of them are implemented and can be mixed with each other.
The low- and high pass filter operations can be expressed as 2D spatial convolutions of an input image $\mathbf{I}$ of size (x_i, y_i) and a (quadratic) filter kernel $\mathbf{K}$ of size (x_k, y_k).

$$C(i,j) = \sum_{m=-a}^{a} \sum_{n=-b}^{b} K(m,n) \cdot I(i-m, j-n) \tag{4.1}$$

where C is the filtered output frame, $a = \lfloor \frac{x_k}{2} \rfloor$ and $b = \lfloor \frac{x_i}{2} \rfloor$. This method is also known as moving window filtering and is supported by opencv using the `filter2D()` method.
The low pass filter especially helps to reduce noise in an input video frame. Because of its' blurring character, the edges around the athletes' body lose their details as well. However, especially in low light conditions, this can be a good first approach to get better results from the key point detector stage. The high pass filter on the other hand is not suitable for videos that contain noisy frames. The the noise would be amplified as well as the edges which leads to a worse body key point detection performance. However, in high quality video recordings, the high pass filter helps to enhance the overall sharpness and therefore especially the edges around the athletes' body.

To reduce the overall noise level in a video frame while preserving edges, a third filter, namely the *bilateral filter*, is implemented. It is defined by the following equation:

$$C(I)_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma s}(\|p-q\|) G_{\sigma r}(I_p - I_q) I_q \tag{4.2}$$

where $C(I)_p$ is the output intensity of the filtered pixel p. G_{σ_s} is a spatial gauss filter which decreases the influence of pixel q with increasing distance between p and q. $G_{\sigma r}$ is a range gauss filter, meaning it decreases the influence of pixel q with increasing intensity difference between p and q. Their intensities are denoted as I_q and I_p respectively. S represents the set of pixels close to p and is determined by a diameter d around each pixel. The normalization factor W_p is given by:

$$W_p = \sum_{q \in S} G_{\sigma s}(\|p-q\|) G_{\sigma r}(I_p - I_q) \tag{4.3}$$

Compared to a simple low pass filter, which is as well implemented as gaussian filter, the bilateral filter takes advantage of not only considering the spatial relation between a pixel and its neighbors but also the relation between their intensity values. This allows a bilateral filter not only to smoothen high frequencies, but to preserve sharp edges at the same time. Opencv offers the built-in method `bilateralFilter()` to apply a bilateral filter to an input video frame. Besides the frame that should be filtered, it also takes σ_s and σ_r as parameters, which define the algorithms' sensitivity to spatial- and intensity changes as well as the diameter d in pixels.
As the filters time complexity depends on the region around each pixel that is taken into account in the gaussian weighting process, it is implemented as first approach to smoothen an input video frame without losing important details in the regions relevant for the body key point detection.

Detecting body key points

Each pre processed video frame is passed to the key point detector stage. This stage is based on the mediapipe pose detection framework (see subsection 3.1.2 and subsection 3.1.3). The models used in this work are the BlazePose models. BlazePose is a convolutional neural network with an architecture similar to MobileNetV2.
It is capable of detecting a total of 33 body key points (shown in Figure 3.1). All of them are tried to be detected in each frame, while only few of them are used in the following parameter calculation process.
The BlazePose network is able to work directly with RGB images as. Thus, the filtered results form the pre-processing stage are as well in the RGB color space. The output generally is a 5-tuple for each detected key point. Besides `x and y` coordinates, BlazePose also offers a `depth estimation` and values describing the probability of a key point being `present` and `visible` in the current frame. The x and y values are normalized according to the image width and height. Furthermore, BlazePose offers three different models[1] which differ in the amount of parameters, thus in the networks' complexity and the resulting detection accuracy. Their inference times range from 0.02s to 0.25s per input video frame respectively. Hence, the least complex model can also be used for real time pose detection. This project supports all three models, as a quick and less accurate analysis can be sufficient (especially in good lighting and recording conditions), whereas the best detection results, especially in poor quality videos are achieved by using the heavy model.
The detected poses can be visualized as can be seen in following Figure 4.3:

[1] light, full and heavy

(a) Sample input frame (b) Pose detection output

Figure 4.3.: Sample pose detection output.
(a) is the input video frame and (b) is the pose detector output. The input frame was pre processed using a high pass filter.

Arm- and knee angle calculation

Based on the previously detected body key points, the parameters shown in Figure 4.1 can be calculated. Especially knee angles during the run up- and takeoff phase can give important insights into an athletes' overall jumping dynamics. In this context, lower knee angles during the run up (at the toe-off point) as well as during the takeoff (swing leg) were found to lead to better jumping results[20]. Thus, these parameters are calculated by the software and can be saved and visualized afterwards.
In the following, the angle calculation is explained based on the knee angle calculation. Taking the relevant detected key points into consideration, following situation is given:

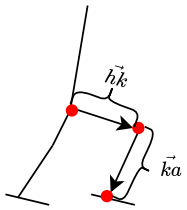


Figure 4.4.: Knee angle calculation and related body key points.
Red dots represent the body key points needed to calculate the knee angle. $\vec{hk}$ and $\vec{ka}$ denote the hip-knee vector and the knee-ankle vector respectively.

The vectors $\vec{hk}$ and $\vec{ka}$ can be calculated using the detected key points' coordinates:

$$\vec{hk} = \begin{bmatrix} knee.x - hip.x \\ knee.y - hip.y \end{bmatrix} \text{ and } \vec{ka} = \begin{bmatrix} ankle.x - knee.x \\ ankle.y - knee.y \end{bmatrix} \tag{4.4}$$

Where .x and .y denote the key points' respective x and y coordinates.
The knee angle θ_{knee} is then represented by the angle between $\vec{hk}$ and $\vec{ka}$ and can be calculated by using the scalar product of $\vec{hk}$ and $\vec{ka}$ and their norms (lengths):

$$\theta_{knee} = \arccos\left(\frac{\vec{hk} \cdot \vec{ka}}{|\vec{hk}| \cdot |\vec{ka}|}\right) \tag{4.5}$$

where $\cdot$ defines the scalar product in this case. The scalar product and the norm are calculated using numpys' `dot()` and `norm()` methods.
The arm angle calculation is performed analogously and is therefore not shown in detail. As the knee- and arm angle calculations have to be done for each video frame, they have to be implemented efficiently not to negatively influence the overall analysis time. Thus, numpy is used to perform the calculations.

Listing 4.1: Angle calculation

```
def calc_angle(first_vec, second_vec):
    first_vec_norm = np.linalg.norm(first_vec)
    second_vec_norm = np.linalg.norm(second_vec)
    if first_vec_norm == 0 or second_vec_norm == 0:
        return np.nan
    return 180 - np.rad2deg(
        np.arccos(
            (np.dot(first_vec, second_vec))
            / (first_vec_norm * second_vec_norm)
        )
    )
```

The above shown `calc_angle()` function takes the two vectors as arguments between which the angle should be calculated. These vectors are, in case of the arm- and knee angles, directly given by the detected body key point positions (see section 4.1.1 and Equation 4.4).
By using this implementation, the overall angle calculation is sufficient fast (performance analysis are shown in subsection 4.1.4) in comparison to the keypoint detection process in order to allow for a quick on-field analysis.

Takeoff angle calculation

As can be seen in Figure 4.1, in the moment of the takeoff, the Center of mass (CM)'s vertical velocity changes its direction rapidly. This change can be quantised by the angle around which the overall velocity vector rotates. The calculated angle is in the following

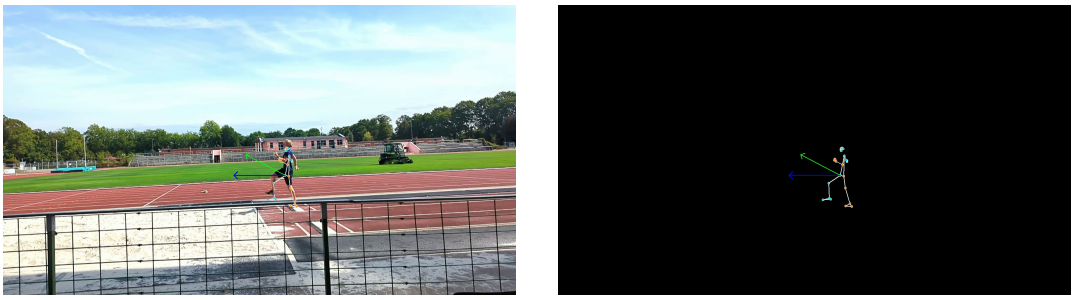
referred to as *takeoff angle* which is one of the most important pre-jump parameters. As shown in [21], even small deviations of around 1° of the optimal takeoff angle can lead to shorter measured distances (see Figure 4.1) of up to 5 cm. The optimum takeoff angle differs between athletes and is especially correlated to the takeoff speed [22].
To calculate the takeoff angle, two vectors are needed. The first vector is a horizontal vector origin in the athletes CM. The second vector represents the athletes'velocity, which, in this case, is considered equal to their CM's velocity. This vector is calculated based on the takeoff frame, which needs to be determined first (see subsection 4.1.2). The second frame which is used to calculate the velocity vector is chosen according to an offset f based on the videos' frame rate[2]. The CM's velocity vector is therefore given by:

$$\vec{v}_t = \frac{1}{f} * (\vec{x}_{t+f} - \vec{x}_t) \tag{4.6}$$

where f is the frame offset and $\vec{x}_i$ are the position vectors. Here, $\vec{x}_t$ represents the CM position at the moment of the takeoff whereas $\vec{x}_{t+f}$ denotes the CM position f frames later. Equation 4.5 can then be applied again yielding following equation for the takeoff angle:

$$\theta_{\text{takeoff}} = \arccos\left(\frac{\vec{v}_t \cdot \vec{h}_t}{|\vec{v}_t| \cdot |\vec{h}_t|}\right) \tag{4.7}$$

where $\vec{h}_t$ denotes any horizontal vector, e.g. $\vec{h}_t = [1, 0]$.
Equation 4.7 is calculated by using the function shown in Listing 4.1 by passing the two vectors shown in following Figure 4.5:



(a) Poses shown as overlay (b) Extracted vectors and positions

Figure 4.5.: Takeoff frame annotated with the two vectors that define the takeoff angle marked green.

The horizontal vector in above Figure 4.5 represents $\vec{h}_t$ in Equation 4.7 and the other vector visualizes $\vec{v}_t$ respectively. Moreover, the velocity vector is scaled by a factor of 20

[2]Chosen as $\frac{1}{10}$th of the frame rate

due to visualization reasons.

4.1.2. Automatic takeoff frame detection

Generally, a long jump can be divided into three different phases, namely approach, jump and landing. In Figure 4.1, an overview of the phases is presented, with the takeoff phase denoted by the first phase transition point (PT_0).
Of all phases, understanding the dynamics of the takeoff phase is especially important, as it represents the last opportunity for an athlete to actively influence critical jumping conditions (e.g. takeoff speed, takeoff angle, ...). Within this short[3] time period the initial kinetic run-up energy is mostly transformed into elastic energy[24].
Moreover, the velocity vector of an athletes' CM changes its direction in the moment of the takeoff[25], as illustrated in Figure 4.1. The forces produced during the takeoff strongly influence the resulting jumping distance[26].
Due to the short time period a takeoff takes, it can be difficult to detect the takeoff frame in a long jump video in order to be able to analyze the exact pre-jump conditions.

However, especially to allow for an on-field jumping analysis, a quick takeoff frame detection is crucial. Thus, an algorithm that can automatically detect the takeoff frame in a long jump video is implemented in the following.

The identification of the takeoff frame relies solely on the video input. This means, the takeoff point has to be derived by analyzing (a combination of) the parameters listed in section 4.1. Additionally, arm angles (and position) are not considered to be of great interest to detect the takeoff in the following. Therefore, the left- and right foot positions, both knee angles and the position of the CM are analyzed in pre-recorded long jump videos of male long jumpers at different professional levels reaching from hobby- to Olympian athlete. The analyzed videos differ in their length and quality as well as in the jumping part they show (e.g. some videos do not show the full run-up). The results of three exemplary analysis are presented in Figure 4.6. The best suitable parameter for the takeoff frame detection is selected based on these analysis results. To compare[4] the various analyzed parameters, the takeoff frame was initially manually selected (highlighted as a blue vertical line in each analysis within Figure 4.6).
The presented data has not been cleaned up in any way. This can especially be seen in the first and last analysis, in which the position and angle data is not accurate due to body

[3] 0.1s - 0.2s[23]
[4] Regarding their suitability as takeoff frame indicators

key point detection inaccuracies. These inaccuracies however are not of great interest at this point as the key points are detected correctly during the approach after a few frames.

In each analysis the left and right foot positions as well as the according knee angles interchange from step to step. The relative CM height remains on the same level during the approach.
Behind the takeoff point, the relative hip- and foot heights increase quickly until the maximal height is reached and the landing phase starts. Moreover, the swing legs' foot height increases faster in comparison to the jumping legs' foot height as the latter one stays on the ground longer to introduce the jump.
In the moment of the takeoff, the knee angle of the jumping leg is above 170 degree, meaning the jumping leg is fully extended. The swing legs' knee angle varies in the shown examples around 100 degree.

In order to reliably detect takeoff frames, an indicator is needed that is suitable for different video inputs and therefore especially insensitive to different run-up and jumping techniques. Taking the results in Figure 4.6 into consideration, it can be noted that all analyzed positions (left-/ right foot and CM) rapidly change their height after the takeoff. However, as the CM's position is more stable in all shown analysis, it is chosen over the foot positions as first takeoff indicator. The second indicator is based on the jumping legs' full extension during the takeoff. Thus, it is chosen as a combination of both knee angles.

The following takeoff frame detection is therefore based on both selected indicators (CM height and knee angles). The primary indicator is chosen as the CM height, while the knee angles are chosen as secondary indicator.
To detect the takeoff based on the primary indicator, a mathematical description of the CM trajectory is modelled using multiple regressions which allows for analytically determining the takeoff point. The secondary indicator is then used to verify the found takeoff frame. In particular, a the takeoff frame is only considered to be correctly identified if the corresponding (jumping leg's) knee angle is above 170°.
The process is explained in detail in the following.

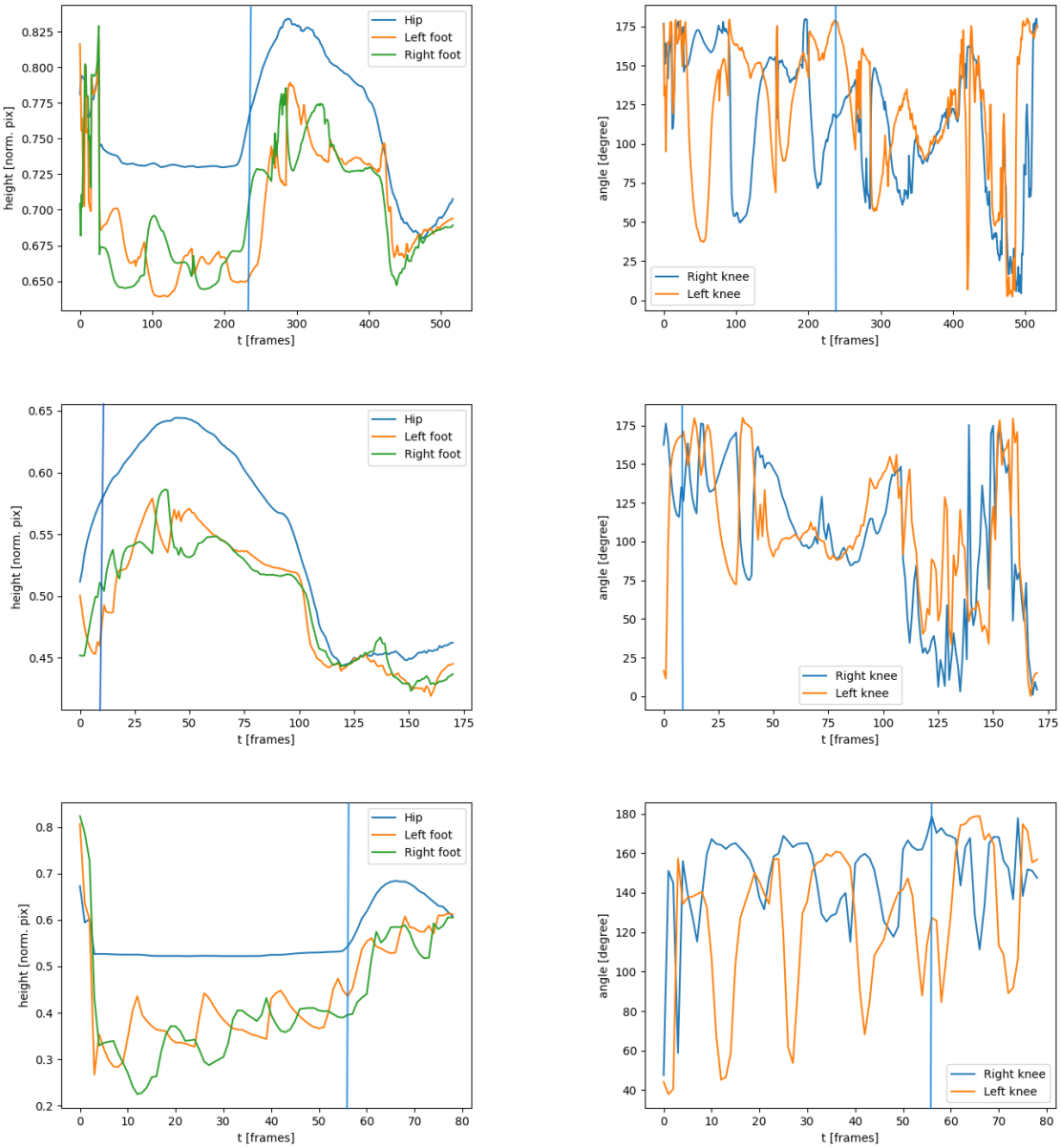


Figure 4.6.: Analyzed and calculated jumping parameters over time.
Left column: Analyzed left / right foot height and hip height.
Right column: Calculated knee angles over time.
The vertical lines represent the visually selected takeoff point.
First row: full run-up, full jump. **Second Row**: short run-up, full jump.
Third row: full run-up, full jump, poor video quality

Regression

As the takeoff point detection is based on linear and quadratic regression, both are briefly introduced in the following. Regression generally describes the approximation of a polynomial function to fit a given dataset. The dataset that is tried to be approximated

in this case is the height of the CM. The dataset can be expressed as (T, H), where H holds the heights of the CM and T holds the related time points. As the input is a video, T represents a simple vector holding the video's frame numbers. For the following steps it is helpful to express T and H as column vectors:

$$\vec{t} = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ n \end{bmatrix} \quad \text{and} \quad \vec{h} = \begin{bmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{bmatrix} \tag{4.8}$$

where n is the total number of video frames and h_i is the $i-th$ recorded height of the CM. Taking the CM in Figure 4.6 into consideration, a whole jump can hardly be fitted with a single polynomial function. However, as shown in Figure 4.1, a long jump consists of multiple phases.
This offers an opportunity to detect the takeoff. As mentioned before, the takeoff frame is defined as the point where the approach phase ends and the jumping phase starts, (PT_0 in Figure 4.1). Thus, if a mathematical expression can be found for each phase respectively, the takeoff frame is found implicitly. The algorithm used for this task is based on the algorithm for a takeoff frame detection developed by Muniz [27].
Other than supposed in his work, the detection algorithm in this work is not based on the foot positions, but on the CM position. There are several reasons for preferring the CM over the foot positions. One of the most important reasons lies in the different jumping techniques, more specifically in the resulting differences during the jumping phase. While the foot path during the jumping phase is rather smooth for athletes using the *hang technique*, it is very different for athletes using the *hitch-kick technique*. Latter one is characterized by quick foot position interchanges, making it more error-prone for inaccuracies in the corresponding body key point detection. Thus, approximating the jumping phase based on the foot positions is more complex and less precise. The CM however follows a similar path independent from the jumping technique, making it more suitable for the given purpose of detecting the takeoff frame.

In the following, the algorithm is explained in detail. All mentioned frame numbers and jumping phases refer to Figure 4.1.
The runup phase includes all frames in the interval $[0, PT_0[$, the jumping phase is covered by the frames in range $[PT_0, PT_1[$ and the landing phase is shown by the frames in the interval $[PT_1, n[$, where PT_0 defines the first Phase Transition point (the takeoff) and PT_1 the point at which the landing phase starts. Each phase by itself can be approximated by a

polynomial function. The approach and the landing can be fitted using a linear expression. Thus, two linear regressions are performed separately to find two expressions of the form

$$y = \beta_0 + \beta_1 t \tag{4.9}$$

which directly yields following linear regression function:

$$y_i = \beta_0 + \beta_1 t_i + \epsilon_i \qquad i = \text{start}, \ \ldots \ , \text{end} \tag{4.10}$$

where β_0 and β_1 are the coefficients that need to be found. start and end represent the first and last frame index that should be considered in the linear regression. For the approach phase this leads to $start = 0$ and $end = TP_0 - 1$, the landing phase is accordingly represented by $start = TP_1$ and $end = n - 1$.
Equation 4.10 can also be expressed as matrix equation:

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_r \end{bmatrix}}_{\vec{y}} = \underbrace{\begin{bmatrix} 1 & t_{\text{start}} \\ 1 & t_{\text{start+1}} \\ \vdots & \vdots \\ 1 & t_{\text{end}} \end{bmatrix}}_{\substack{\text{Design matrix} \\ T}} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}}_{\substack{\text{Coefficients} \\ \text{vector} \\ \vec{\beta}}} + \underbrace{\begin{bmatrix} \epsilon_0 \\ \epsilon_1 \end{bmatrix}}_{\substack{\text{Error} \\ \text{vector} \\ \vec{\epsilon}}} \tag{4.11}$$

where $r = end - start$. This can be written in short form as:

$$\vec{y} = T\vec{\beta} + \vec{\epsilon}$$

Now, the vector $\vec{\beta}$ needs to be found that minimizes the sum of errors E_{SSE} between the measured CM heights $h_i \in \vec{h}$ and the fitted polynomial y_i at time step i. To measure the error, the Sum of Squared Erros (SSE) is used:

$$E_{SSE} = \sum_{i=start}^{end} \epsilon_i^2 = \sum_{i=start}^{end} (h_i - y_i)^2 \tag{4.12}$$

Using the simple linear expression from Equation 4.10, following error function is found:

$$E_{SSE} = \sum_{i=start}^{end} (h_i - \beta_0 - \beta_1 i)^2 \tag{4.13}$$

By using the matrix notation introduced in Equation 4.11, the equation above can be written in matrix notation as well:

$$E_{SSE} = (T\vec{\beta} - \vec{h})^T (T\vec{\beta} - \vec{h}) \tag{4.14}$$

As $\vec{\beta}$ should minimize E_{SSE}, the minimum of E_{SSE} is needed, which can be found by solving following equation:

$$\nabla_{\vec{\beta}} E_{SSE} = \nabla_{\vec{\beta}} (T\vec{\beta} - \vec{h})^T (T\vec{\beta} - \vec{h}) = 0 \tag{4.15}$$

where $\nabla_{\vec{\beta}}$ denotes the gradient with respect to $\vec{\beta}$.
The coefficients can then be found by solving Equation 4.15 for $\vec{\beta}$, which yields the following normal equation:

$$\vec{\beta} = (T^T T)^{-1} (T^T \vec{h}) \tag{4.16}$$

that especially requires $(T^T T)$ to be invertible. A proof of Equation 4.16 can be found in [28].
As mentioned, this process is performed twice to find a linear approximation for the approach and the landing phase of an athletes' CM respectively.

The jumping phase cannot be approximated with a linear polynomial. However, as can be seen in Figure 4.6 after the marked takeoff point, the curve of the CM height can be approximated using a parabola. Thus, a quadratic regression is used to find a curve that describes the height of the CM during the jumping phase. The second order polynomial that needs to be found is of the form:

$$y = \beta_0 + \beta_1 t + \beta_2 t^2 \tag{4.17}$$

yielding the quadratic regression function:

$$y_i = \beta_0 + \beta_1 t_i + \beta_2 t_i^2 + \epsilon_i \qquad i = PT_0, \ \ldots \ , PT_1 - 1 \tag{4.18}$$

PT_0 and PT_1 are the phase transition points shown in Figure 4.1. The following steps are equal to the linear regression shown above. Thus, only the differences are shown in detail. The T matrix in Equation 4.11 contains all frame numbers as column vector. Because a linear relation was tried to be found to approximate the approach and landing phase before, the T matrix as well only contained linear values. Now however, a quadratic

relation needs to be found. Thus, the T matrix needs to be extended by one more column holding the quadratic frame numbers yielding following matrix equation:

$$\underbrace{\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_r \end{bmatrix}}_{\vec{y}} = \underbrace{\begin{bmatrix} 1 & t_{PT_0} & t_{PT_0}^2 \\ 1 & t_{PT_0+1} & t_{PT_0+1}^2 \\ \vdots & \vdots & \vdots \\ 1 & t_{PT_1-1} & t_{PT_1-1}^2 \end{bmatrix}}_{T} \underbrace{\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \end{bmatrix}}_{\vec{\beta}} + \underbrace{\begin{bmatrix} \epsilon_0 \\ \epsilon_1 \\ \epsilon_2 \end{bmatrix}}_{\vec{\epsilon}} \tag{4.19}$$

where $r = PT_1 - 1 - PT_0$. The error function which needs to be minimized can be set equivalent to the linear case (Equation 4.14). Comparing Equation 4.11 and Equation 4.19 the only things that change are the number of coefficients (and thus the number of error terms) as well as the design matrix T, which holds one more column. The steps to determine the coefficients $\vec{\beta}$ are equal to the linear case in equations 4.14 to 4.16. $\vec{\beta}$ is then given by the same normal equation 4.16 as in the linear case:

$$\vec{\beta} = (T^T T)^{-1}(T^T \vec{h})$$

where $\vec{\beta}$ contains three coefficients β_0, β_1 and β_2 to approximate the jumping phase.

By using this linear- and quadratic regression approach, the whole jump can be modelled mathematically. However, in order to find the best fitting model, the phase changing points (PT_0 and PT_1 in Figure 4.1) need to be determined. This can be done by minimizing the overall error E_{total}, which is defined as the sum of regression errors resulting from the three independent regressions performed (one per jumping phase):

$$E_{total} = E_{approach} + E_{jump} + E_{landing} \tag{4.20}$$

E_{total} can be minimized by using a brute-force approach. For each possible combination of phase transition points, two linear regressions (representing approach and landing) and one quadratic regression (representing the jumping phase) are performed and their individual errors given by Equation 4.14 are summed up. The combination of phase transition points that leads to the minimal E_{total} is then considered as the optimal model to fit the overall jump. As the found phase transition points directly represent their corresponding frame numbers, the takeoff frame is directly given by the first phase transition point.
The described process is outlined in following Algorithm 1.

Algorithm 1 takeoff_frame(cm_height: array, knee_angles: array)

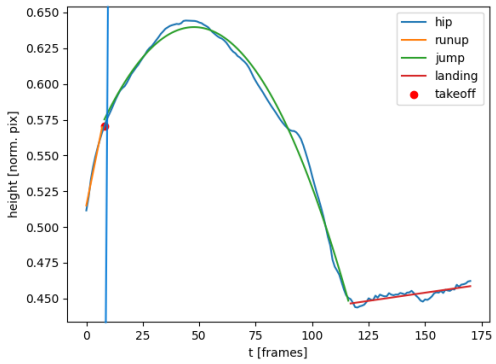
```
n ← length of cm_height
total_error ← initial error
transition_points ← (0, 0)
for i ← 0 to n − 1 do
    for j ← i + 1 to n do
        β⃗_approach, error_approach ← lin_reg(hip_height[: i])
        β⃗_jump, error_jump ← quad_reg(hip_height[i : j])
        β⃗_landing, error_landing ← lin_reg(hip_height[j :])
        fitting_error ← error_approach + error_jump + error_landing
        if fitting_error < total_error and β⃗_jump[0] < 0 and knee_angles[i] > 170 then
            total_error ← fitting_error
            transition_points ← (i, j)
        end if
    end for
end for
return transition_points
```

The expression $\vec{\beta_{jump}}[0] < 0$ guarantees, that the found parabola for approximating the jumping phase is opened downwards. Moreover, $knee\_angles[i] > 170$ makes sure, that only frames in which the knee angle is above 170 degrees can be considered as takeoff point.

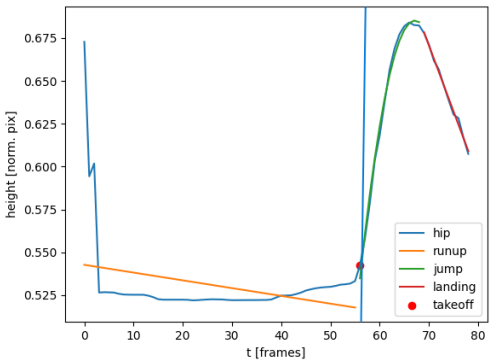
The shown algorithms' runtime scales according to $\mathcal{O}(n^2)$ where n is the number of video frames [5]. Thus, it is not suitable for long video sequences. However, as a complete long jump takes only a few seconds, the algorithm can be used for an on-field jump analysis. Algorithm 1 was then applied to all three datasets shown in Figure 4.6. The results are shown in Figure 4.7.

For better understanding, the height of the CM was extracted from the datasets as it is the only parameter relevant for the takeoff detection. The data is not cleaned up in any way, thus, outliers are still visible. However, as can be seen, the automatically detected takeoff frames are similar to the manually annotated frame numbers.

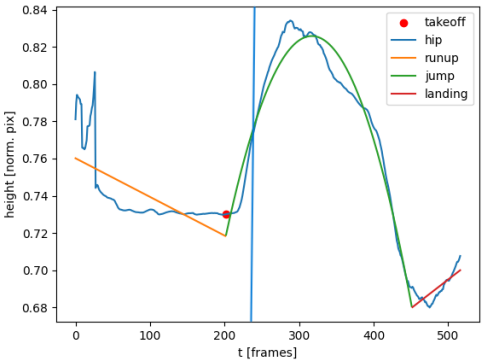
[5] in which body key points were found



(a) Short approach, reliable keypoint quality



(b) Long approach, outliers in approach



(c) Long approach, outliers in approach and jump

Figure 4.7.: All three jumping phases approximated using linear regression (runup and landing) and quadratic regression (jumping phase).
The automatically detected takeoff frame is marked with a red dot.
The vertical blue lines represent the manually marked takeoff frames.

4.1.3. Saving analysis results

After a complete jump analysis has been performed, all data, especially the calculated parameters such as arm- and knee angles need to be stored permanently. Thereby, each input video file must only be analyzed once in order to avoid costly double analysis. While the detected body key points are stored in form of an annotated video file, the analysis parameters are stored alongside in a separate file which can be re-loaded and visualized. This file will in the following be referred to as `Parameter file`.
As most of the parameters are calculated frame-wise (like knee angles, foot positions, ...), the well-structured hdf5 file format (see subsection 3.1.5) is used for this task. The exact file structure is shown in following Figure 4.8:

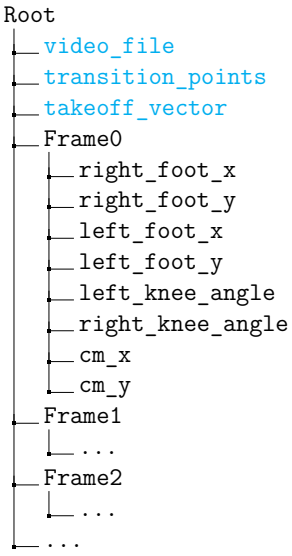


Figure 4.8.: HDF5 Parameter file structure

As can be seen in above Figure 4.8, each hdf5 file contains three metadata parameters. They do not belong to a certain frame and represent important analysis results that take several frames into consideration. Furthermore, these parameters can be accessed efficiently. The `transition_points` metadata contains the phase transition points (see Figure 4.1), whereas `takeoff_vector` is a tuple of the form $(\theta_{\text{takeoff}}, \vec{v}_t)$ according to Equation 4.7. As one of the shown Parameter files is created per jump analysis, an annotated video file belongs to each of those files. Thus, a connection to the related video file is saved in the Parameter file in form of the `video_file` metadata.
Furthermore, there is a group created for each analyzed video frame. Within these groups there is one dataset created for each analysis parameter.

To allow simple and efficient read- and write operations, the class `Parameterfile` is implemented, which abstracts the described parameter file. All HDF5 file interactions are implemented using the python h5py package[29], which offers dedicated methods for creating a whole HDF5 file as well as methods for creating groups, datasets and metadata. Taking into consideration that, for each input video frame, one group in the according HDF5 file as well as one annotated video frame needs to be saved, there are many memory accesses during the analysis process, which leads to poor runtime behavior. To reduce the number of memory accesses, the Parameterfile instances buffer some HDF5 frame groups and then perform batch saving. This effectively reduces the number of costly memory accesses, leading to a better performance behavior. The software uses a batch size of 128 by default.

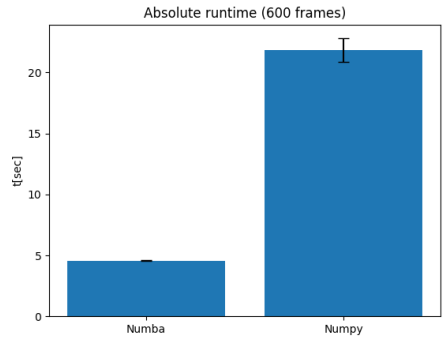
Overall, the `Parameterfile` class manages all analysis file interactions. Possible interactions can be seen in its class diagram:

Figure 4.9.: Parameterfile class diagram

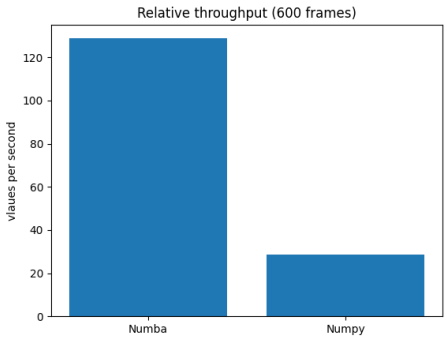
In above Figure 4.9 the `load()` method and all `get_x()` methods are used to retrieve analysis parameters from saved Parameterfiles.

4.1.4. Runtime considerations

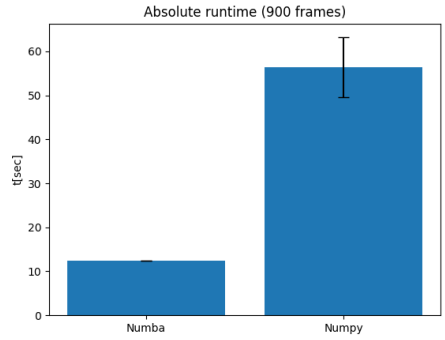
The software is mainly developed towards a high key point detection and thus, a high parameter calculation accuracy. However, as it is meant to be used as an on-field analysis tool, quick analysis times are important. Hence, in the following, some important run times are shown and discussed.
All following evaluations were performed on a laptop equipped with an AMD Ryzen 5800U processor and 16GB of ram. This setup was chosen to represent a standard setup that can be used for an on-field analysis.
Generally, there are two main runtime critical parts in the jump analysis process: first, the key point detection and second, the takeoff frame detection. The first one mainly depends on the users' choice of the underlying key point detection model (evaluated later). The latter one however performs many numerical operations. As a result, it can potentially profit from pre-compiled implementations using `numba` (see subsection 3.1.6).
To validate this assumption some runtime comparisons based on video inputs of 10 seconds length are performed. One is recorded at 60 Frames Per Second (FPS) and the other one at 90 FPS, resulting in 600 and 900 frames respectively. The results are presented in following Figure 4.10:



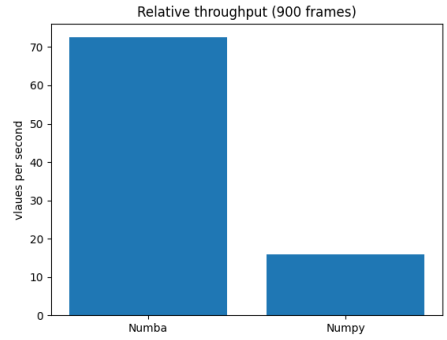
(a) Absolute runtime comparison. Error bars represent the standard deviation.



(b) Normalized throughput comparison. Values are directly proportional to the runtime shown in (a).



(c) Absolute runtime comparison.



(d) Normalized throughput comparison.

Figure 4.10.: Runtime and throughput comparison between pre-compiled (`numba`) and non pre-compiled takeoff frame detection implementations.

As can be seen, the takeoff frame detection generally scales according to $\mathcal{O}(n^2)$, where n is the number of analyzed video frames (explained in subsection 4.1.2). Looking at a 900 frame video, this leads to $900 * 900 = 810 * 10^3$ loop runs. Moreover, in each iteration, three polynomial regressions are performed which leads to around $2.4 * 10^6$ regression operations. Due to that many regression operations, the two evaluated implementations differ in the way they perform the necessary regressions (see Algorithm 4.1 for details). While the python implementation relies on `numpy`'s polyfit[6] method, the pre-compiled `numba` implementation uses a custom regression implementation which is set up according to equations 4.11 to 4.16. A custom implementation is chosen, as the polyfit method offered by `numpy` cannot be pre-compiled using `numba`. The complete implementation is not shown, however, the main function is presented in order to demonstrate the (simple) usage of `numba`:

[6] `https://numpy.org/doc/stable/reference/generated/numpy.polyfit.html`

Listing 4.2: Polynomial regression

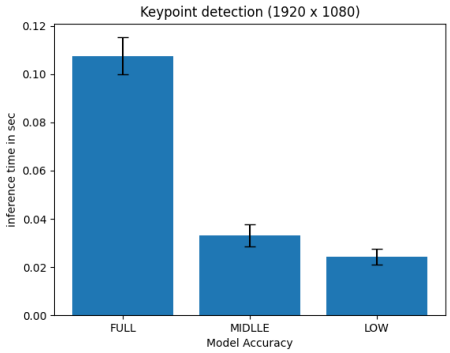
```
import numba as nb

@nb.jit('f4[:](f4[:], f4[:], i4)') # numba decorator
def _fit_poly(x: np.ndarray, y: np.ndarray, deg: int):
    A = _coeff_mat(x, deg) # creates design matrix
    p = _fit_x(A, y) # solves Ax = y
    return p[::-1] # ensure p[0] to be the highest order
       coefficient
```

The methods `_coeff_mat` and `_fit_x` represent Equation 4.11 and Equation 4.16 respectively. `_fit_x` uses the least-squares fitting approach shown in Equation 4.15 to find the best fitting model. Furthermore, the residuals (errors) which are needed (see Equation 4.20) are calculated via polynomial evaluation using Horner's method [30].
As can be seen, the only difference to a standard python function definition is the `nb.jit()` `decorator` which wraps the `_fit_poly` function in `numba`s' jit function to pre-compile it when the function is first called (therefore the name *Just-in-time compiler*).
This behavior can potentially lead to a large execution overhead before the function can be executed for the first time. However, numba can also pre-compile the function when the application is launched to avoid long compiling processes during runtime. This is achieved by passing the parameter types to numba (`'f4[:](f4[:], f4[:], i4)'`), where f4 denotes a standard 32 bit single precision float, i4 a standard 32 bit integer value and [:] their array types respectively. Moreover, the first `f4[:]` defines the functions' return type as an array of 32 bit floats.
Numba needs that extra information to pre-compile the function, as the functions' parameter types cannot be derived just by the function definition, as python generally is a dynamically typed language.

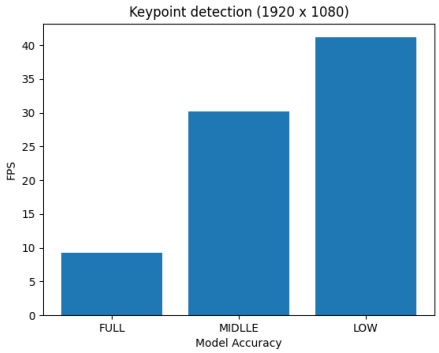
Taking the implementations and Figure 4.10 into account, the takeoff frame detection significantly profits from the pre -compiled numba implementation. The absolute runtime is reduced by over 80%, increasing the throughout by a factor of around 5.5. Thus, the final takeoff frame detection is implemented using `numba`s' JIT compiler.

The other runtime crucial part is the body keypoint detection. As aforementioned, mediapipe offers three different BlazePose keypoint detection models differing in terms of accuracy and performance. All three models are implemented, thus their inference times

are compared. As the final analysis will be performed on videos of size 1920 x 1080, the performance is tested on video frames of this size. The results are presented in following:



(a) Absolute inference time comparison. Error bars represent the standard deviation.



(b) Theoretically achievable FPS comparison. Values are directly proportional to the runtime shown in (a).

Figure 4.11.: Inference time comparison of all three mediapipe pose detection models.

As can be seen in above Figure 4.11 the inference times significantly differ between the three BlazePose pose detection models. While the full, thus most complex and most accurate detection model achieves speeds of up to around 10 FPS, the light model can evaluate up to 40 FPS. This means latter one can theoretically even be used as a real time analysis option for video recordings at 30 FPS. However, as this work focuses on the video analysis, the real time capabilities are not part of this works' scope.
The parameter calculation (see section 4.1) is neglected in the runtime measurements, as they are much less computational expensive compared to the body key point detection and the takeoff frame detection.
Thus, a full long jump video of 10 sec lengths (60 FPS) can be fully analyzed in around a minute.

4.1.5. Visualization in a GUI

To present the jump analysis, a GUI is implemented. This GUI is mainly developed regarding three purposes:

1. Choose video files and analysis parameters.
2. Present analysis results while the analysis is running.
3. Present the fully analyzed video and the calculated parameters.

In the following, the GUI will be shown and explained in two steps. At first, the first two points in above listing will be shown and secondly, the full analysis visualization is presented.

The second point allows an athlete (or trainer) to analyze *single frame parameters* while the video is still being evaluated. Single frame parameters are those, that do not need all frames to be evaluated but just rely on the current frame (e.g. knee angles and arm angles). In contrast a full analysis includes all other parameters, such as the takeoff frame and the takeoff angle. This allows to reduce the actual on-field analysis times, as a first overall jumping impression can be offered immediately.
The GUI that is used for the first and second point (see above) is presented below:

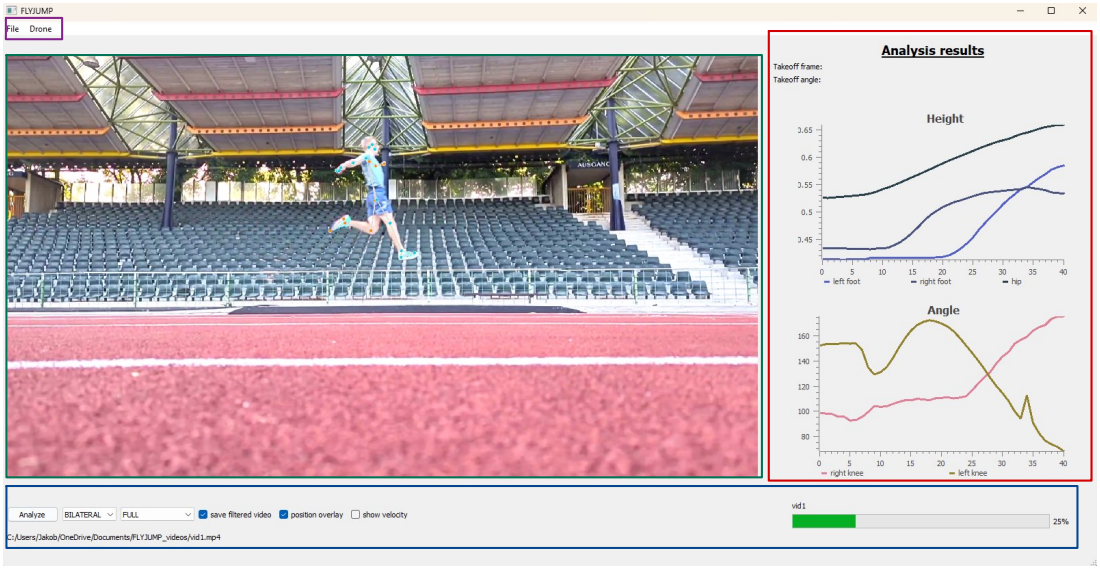


Figure 4.12.: GUI shown while the video analysis is running.

As can be seen, the GUI is split into three main parts. These are marked red, green and blue respectively in above Figure 4.12.
The blue area is used to let the user choose important analysis parameters and show the analysis progress. The parameters are the filter function to be used (see section 4.1.1) as well as the key point detection accuracy (see subsection 3.1.2) and the output mode. Moreover, the athlete can choose to get the velocity vector calculated and visualized in each frame.

Above this *analysis control panel*, there is the actual video widget, which is used to display the current video according to its current analysis progress. All analysis results, that the user chose (in the analysis control panel) are already visualized in this stage. Particularly, the detected position overlay (the body key points) are shown. Thus, a first evaluation can be performed while the analysis is still running.

On the right-hand side (red box), the actual analysis results are visualized. In this

stage, only single frame parameters are calculated and shown. More specifically, these are knee angles, left- / right foot positions as well as the hip height. By updating these curves successively for each analyzed frame, an overview of the overall jumping progression is visualized over time.

The GUI changes its layout slightly once the analysis is finished to offer a more convenient way to interact with the analysis results and their respective Parameter files (see subsection 4.1.3). The GUI layout for finished video analysis is presented in following figure:

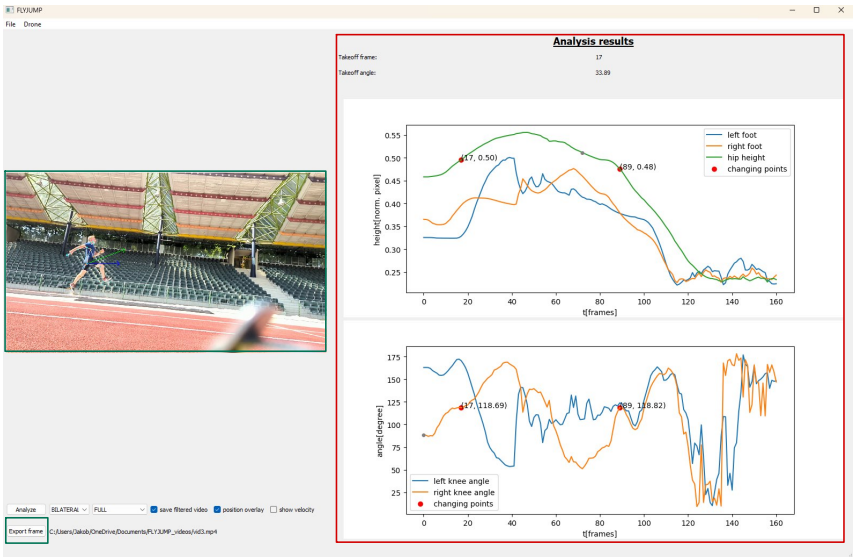


Figure 4.13.: GUI shown once the video analysis is finished or a completed analysis and its corresponding parameter file is loaded.

As can be seen, the analyzed video is presented in the main window (green box in above figure). Here, either the key points are shown as overlay over the video, or they are shown on a black background to emphasize the posture.
Compared to Figure 4.12 mainly the result area changes its layout. Especially, the full video parameters (takeoff frame- and angle) are shown. Moreover, the analyzed video playback starts with the automatically detected takeoff frame on which the calculated takeoff angle (and its according vectors) is annotated.
Furthermore, the analysis area now shows the complete knee angle, foot positions and hip position plots created via the `matplotlib` library. Within these plots the frames corresponding to the phase transition points (see Figure 4.1) are marked red. Additionally, the currently visible frame is marked, too. This ensures the user to be able to visually keep track of the jumping stage shown in the current frame. If hovered over plots, the full values of the plots are shown at this exact frame number. Once the user clicks in a plot, the video playback jumps to the selected frame number. This way, the athlete can jump

through important parts of their total jump according to the analyzed parameters.
Furthermore, the software allows exporting single video frames as images to offer the athlete quick later access to important video frames (such as the takeoff frame).

To allow for a convenient GUI interaction while a video is being analyzed or their results are presented, the video is implemented as an own `Video` class that inherits from PyQt's `QRunnable` class.
Objects of this class can generally be used to either start the analysis process or the video playback. As the class inherits from `QRunnable`, the analysis (and the video playback respectively) can run in a background thread which allows to keep the GUI responsive. Furthermore, multiple video objects can theoretically run concurrently[7] by using PyQt's `QThreadPool`, allowing especially to perform multiple video analysis in parallel. Test runs have shown that two CPU cores per video analysis lead to the best results when performing multiple analysis simultaneously.
As the video is running in a background thread, it can not directly show its' frames in the GUIs' main application thread. Thus, to visualize the video frames, PyQt's signal and slot mechanisms are used. Whenever a video frame is ready to be visualized, the corresponding video object emits a `visualize_frame(Frame)` signal, where `Frame` is a custom class that abstracts a numpy video Frame object. The main thread has a slot to which this signal is connected. This way, video frames are visualized by the main thread only while the background threads are only responsible for the analysis and video loading. The overhead introduced by using signals and slots is not considered too large, as the `Frame` objects are passed by reference (meaning they are not copied).
The interaction with the background video object is realized similarly using signals being emitted by the main thread (e.g. abort analysis, stop playback, jump to a specific frame,...) and corresponding slots executing the actual action in the `Video` class.

4.2. Drone setup

In order to capture high-quality video recordings that cover a complete long jump, from the first step all the way to the landing, a drone is used to fly next to the athlete throughout the whole process. Thus, a drone in form of a quadcopter is built from scratch. Its control will be integrated seamlessly in the projects' GUI.
This section introduces the hardware components that are used for building this drone as

[7] In reality, this is limited by the number of available CPU cores and especially by Pythons' Global Interpreter Lock (GIL)

well as its flight control unit.
A short outline of the hardware is given in subsection 4.2.1, while subsection 4.2.2 focuses on the overall assembly of the selected hardware.

4.2.1. Hardware selection

Currently, commercial drone hardware on the market is mainly separable into the two large areas of fully remote controlled FPV hardware and hardware for (autonomous) drones that can usually carry more load, e.g. heavy cameras. Even though the quadcopter in this project needs to be remotely controllable from a ground station pc, it is still more likely to be located in the latter one.
Generally the hardware was chosen based on the following criteria:

- price
- compatibility
- size

Flight Hardware

The main hardware that a quadcopter needs to fly will, in the following, be referred to as *flight hardware*. This includes frame, motors, rotors, Electronic Speed Controls (ESCs) and a Power Delivery board (PDB).
The main platform on which all drone hardware is mounted, is referred to as a quadcopter's frame. As this project's drone does not need to carry any heavy load, such as high precision camera systems or other sensors, a rather compact frame would theoretically be sufficient. However, compact frames tend to be less stable compared to larger frame sizes which could lead to a lower video recording quality and thus require more complex post-processing software. Moreover, the assembly process on larger frames is more convenient and replacing parts is easier. Additionally, compact frames are most commonly used in areas that demand quick reaction times for high speed flight maneuvers, e.g. in drone racing. This however is not needed in this project's context.
Taken the mentioned considerations into account the mid-sizes *Holybro S500 V2* frame kit is chosen. Besides the frame, the kit also includes a landing gear and rotors. Moreover, the main platform includes a PDB to split the battery's power equally to all four motors. An overview of all included parts is given in Figure 4.14.

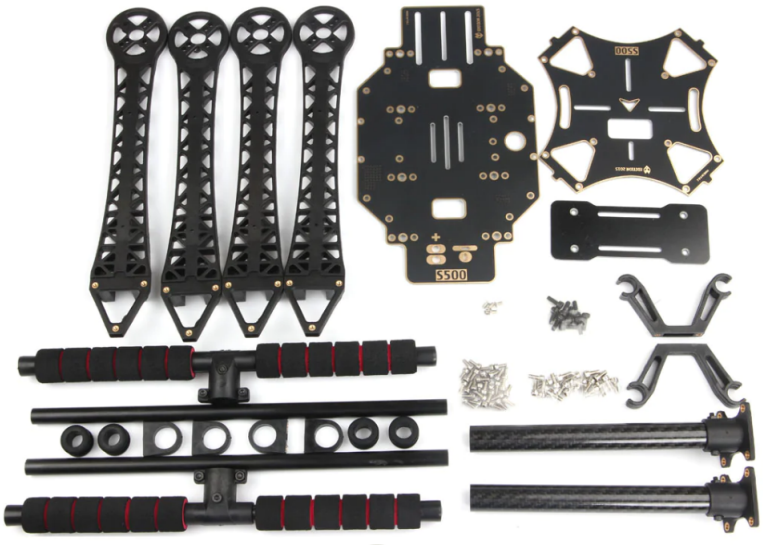


Figure 4.14.: Holybro S500 V2 frame kit

Besides the frame, motors and compatible ESCs are crucial flight hardware components. Each motor requires an own ESC that translates signals from a flight control unit to a voltage and thereby control the motors' rotation speed. To guarantee compatibility, both components were chosen from Holybro as well and can be seen in Figure 4.15.

(a) 920KV Motor (b) Electronic Speed Control

Figure 4.15.: Motor (a) and ESC (b)

The drones' motors performance capabilities are defined by the number of Revolutions Per Minute (RPM) they can perform per 1V input. As can be seen in Figure 4.15a, this link between rotation speed and input voltage is expressed in the arbitrary unit *KV*. The chosen motors are capable of rotating with a speed of 920 RPM per 1V input voltage. Put into context, this is a common rotation speed in commercial and hobby drone applications. Racing drones however, operate at motor speeds of up to 3500 KV.

Control Hardware

In order to perform flight maneuvers with a quadcopter, each motor must be controllable individually. The calculation of the correct rotation speeds is generally performed by a *flight control unit*. Usually, it receives directional instructions from a remote control as input, combines them with many parameters (e.g. GPS position, height over ground, speed, etc.) and generates a (PWM) output signal for each motor.
Within this project the flight controller needs to deal with two different inputs. First, the ground station which can be seen as a remote control in this case. Additionally, the drone should be able to fly autonomously next to an athlete during their long jump training. Here, the second input gets important. The autonomous fly option requires the quadcopter to perform a person detection and therefore image processing on-board. As the flight controller itself is not able to perform such calculations, an additional *companion computer* is required. This companion computer will then send directional instructions just like the ones from the ground station to the flight controller and thereby control the drone.
The combination of flight controller and on-board companion computer will in the following be called *control hardware*.
There are many types of different flight controllers available commercially. However, most of them are not meant to be used in combination with a companion computer.
Two of the most commonly used flight controllers in autonomous drone projects are the *PixHawk* and the *Navio2*. They are often chosen because they both work together seamlessly with a companion computer. The former is a totally independent system which can also operate without any supporting computer. The latter is implemented as Hardware Attached on Top (HAT) specifically designed for a Raspberry Pi. Thus, it does not include an own CPU but uses the Raspberry Pis's resources to perform flight relevant calculations. A detailed comparison between both flight controllers is given in Table 4.1.
As can be seen, both flight controllers offer different interfaces to connect additional hardware. Moreover, both systems include sensors, mostly to gather information about drone's current position and inertia. Here, the Navio2 even offers more sensors, as it already includes a Global Positioning System (GPS) sensor, while the PixHawk relies on an external one.

For this project, the PixHawk was chosen over the Navio2 mainly for three reasons. First, it is on the market for a long time already and thus have a large community support. Secondly, as it is an independent system, a failure of the companion computer will not lead to a crash. Lastly, it allows for a wide range of companion computers, while the Navio2 can only interoperate with a Raspberry Pi.

Flight Controller Comparison		
Criteria	PixHawk	Navio2
Processor	ARM Cortex M4 with FPU / 32-bit co-processor	Depends on Raspberry Pi version
Sensors	ST Micro 16-bti gyroscope, ST Micro 14-bit accelerometer, MEAS barometer	MPU9250 9DOF IMU, LSM9DS1 9DOF IMU, MS5611 Barometer, U-blox M8N Glonass/GPS/Beidou
Interfaces	UART, Spektrum DSM, PPM / S.BUS input, I2C, SPI, CAN, USB, 3.3V and 6.6V ADC input, 8 PWM outputs, 6 Auxiliary outputs	UART, I2C, ADC, PPM / S.BUS input, 14 PWM outputs
Dimensions (W x H x L) in mm	$50 \times 15.5 \times 81.5$	55×65
Other	Failsafe options (e.g. extra power supply, GPS, etc.)	None
Price (Eur.)	TBD	TBD

Table 4.1.: Comparison between PixHawk and Navio2 flight controller.

Furthermore, the mentioned considerations lead to easy rapid prototyping approaches, as the drone can be manually flown without a companion computer in a first implementation.

4.2.2. Hardware assembly

In the following, a high-level overview of the quadcopters' hardware setup is given. The general wiring is shown and explained before a short introduction of some important communication protocols is given.

General wiring

In the following Figure 4.16 the general wiring layout is shown. The whole system is powered from one power source only.

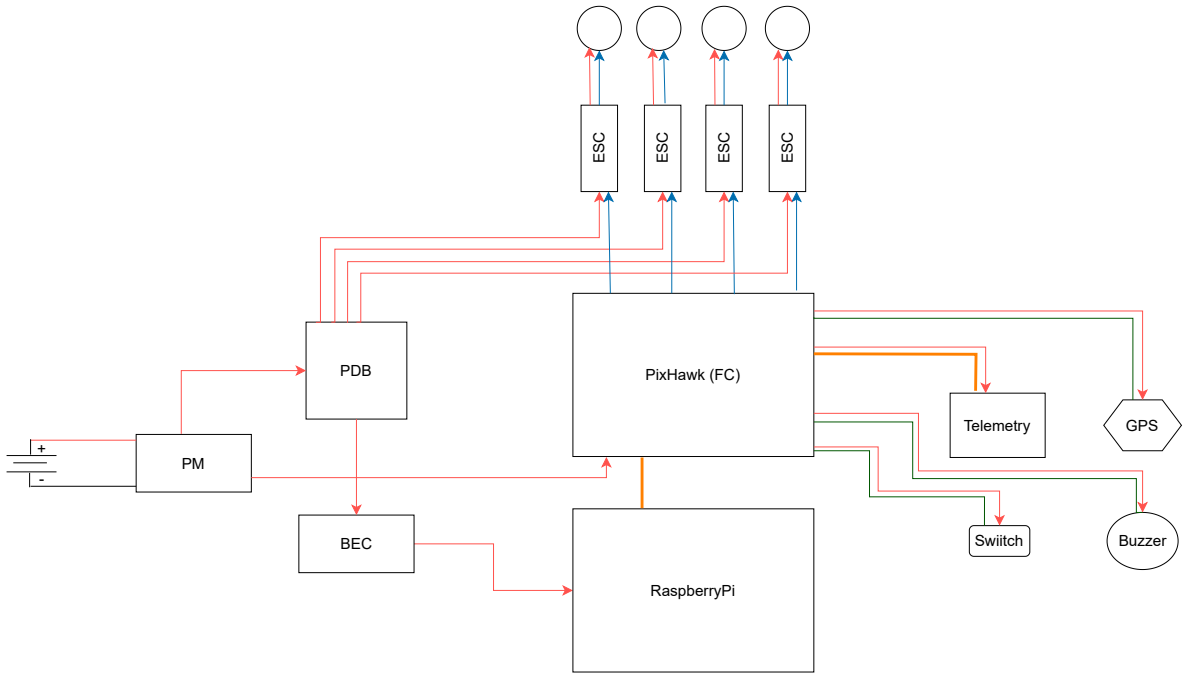


Figure 4.16.: General wiring of the hardware setup.
Power connections are labeled red, Blue connections are used for Pulse Width Modulation (PWM) signals, Green connections are serial connections, orange connections are more specifically serial UART connections.

This results in some challenges in providing the correct voltage for each connected device. In the current setup this task is taken over by three devices. The Power Module (PM) is directly connected to the battery. It transfers the battery's voltage to the PDB and a lower 5V voltage to the PixHawk flight controller. The PDB itself is a parallel circuit, thus providing the same voltage (battery voltage) to each output. The third device is a Battery Elimination Circuit (BEC) which is directly connected to the PDB and delivers a constant 5V output. This can be used to power a companion computer such as a RaspberryPi.
All other required peripherals are powered by the PixHawk flight controller itself. The main peripherals used in this project are a telemetry module which is used for communication with a ground station and a GPS module used for improving the drones capabilities to follow a defined trajectory, which is specifically useful for auto-return and landing features. Two more peripherals, a buzzer to output audio warning signals and a manual kill switch which can immediately stop all four motors, are installed mainly for safety reasons.
The hardware components that actually control the motor rotation speeds, the ESCs, are connected to the PDB for power supply as well as to the flight controller that calculates the correct rotation speeds based on the wanted flight maneuvers and outputs a PWM signal for each motor.
The presented overall wiring is rather complex but allows relying on one power source only

instead of using multiple power sources for flight hardware and control hardware including peripherals respectively.

Communication between components

The drone setup needs hardware components to communicate with each other in order to transfer control signals from either the companion computer or from the ground station to the flight controller. Even if both options origin from different sources, they use the same device-to-device communication protocol. The protocol used for this purpose is the *UART* protocol, which is acronym for universal asynchronous receiver / transmitter protocol. It is based on a serial, full duplex connection using six connections. The connection layout is shown in Figure 4.17,

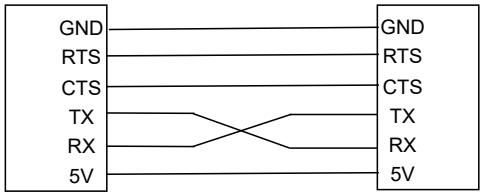


Figure 4.17.: UART wiring

where RTS / CTS denote Ready to send and Clear to send. RX / TX represent the actual data receiving and sending connections respectively.

UART transfers data using data frames with minimal overhead. A typical UART frame consists of just a start bit, data bits, a parity bit and a stop bit. Figure 4.18 visualizes such a data frame.

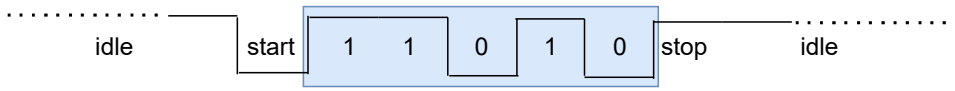


Figure 4.18.: Exmple of an UART data frame. The blue marked area is the actual data part that is transmitted.

The shown UART data frame includes 5 data bits, however, the amount can vary between 5 and 9. Moreover, the included even parity bit is optional. The idle state is set to a voltage that represents HIGH level on purpose, so that any connection failure is easily detectable. UART can work with any voltages to denote HIGH and LOW levels. In the quadcopter setup HIGH is represented by 5V and LOW by GND.

4.2.3. Recording and streaming videos

As the drone is meant to be used to support the long jump analysis process by recording the jump and sending the captured video to a ground station, a complete video recording process is developed and implemented.
This includes the hardware camera setup on the drone, as well as the communication between the ground station which initializes the video recording and the drone capturing the actual video.
Generally, the on-board RaspberryPi takes care of capturing the video recordings. It starts the video recording when it is triggered via the ground station and sends the recorded video file back to the ground station after the recording has ended. The video is then analyzed on the ground station. Moreover, the RaspberryPi offers a live stream of the current camera image to allow for aligning the drone correctly relative to the athlete.
In the following, the implementations of the aforementioned features are shown in detail.

Connection between the ground station and the on-board computer

In order to start or stop video recordings or capture images as well as to send video recordings back to the ground station, the on-board mounted RaspberryPi needs to communicate with the ground station. Furthermore, this connection is used to stream live images from the drone to the ground station, too.
There are many implementations that use analog [31] video transmission for live-streaming videos from FPV drones. More recently, digital alternatives [32] with comparable latencies were developed. However, the live stream in this work is meant to be used for aligning the drone only, not for controlling it. Thus, the real-time requirements are less important. Furthermore, the connection is not only used to live-stream videos, but also to send commands (e.g. start video recording) and video files. Hence, the connection to the on-board computer is realized using a Wifi network. This is achieved by setting up the RaspberryPi as access point to which the ground station can then connect directly.
This creates an arbitrarily useable connection which allows for quick transmission speeds. As the ground station and the drone are always used together on-field (which guarantees physical proximity), the limited Wifi transmission range is not considered as problematic for the following Implementations.

TCP sockets for communication

Even though the video recording is performed by the on-board RaspberryPi, the recording must be initializable by the user via the ground station. Therefore, the communication between these two computers is established using a Transmission Control Protocol (TCP) [33]

connection. As potentially large video files are transferred wirelessly from the RaspberryPi to the ground station, a connection-orientated and confirmed TCP connection protocol is chosen over a connectionless User Datagram Protocol (UDP) [34] to avoid data loss.

When the ground station connects to the drone, a TCP socket is opened in a background thread. The corresponding TCP connection is initialized by the ground station using a three-way handshake[8]. To create the socket, Python's `socket` module is used to access the BSD socket interface. Generally, a TCP socket is uniquely defined by an IP-Address and the corresponding port number. As the ground station is directly connected to the RaspberryPi's own WLAN, the socket's IP address is equivalent to the ground station's standard gateway. The port is set arbitrarily.
To start / end video recordings, pre-defined messages are sent via the created TCP socket. The messages are utf-8 encoded and sent using Python's `socket.sendAll(message)` function. The drone on the other connection end uses `socket.recv()` to listen to any messages sent via the TCP connection. If a message contains one of the pre-defined keywords (e.g. *start_recording*, *end_recording*), the corresponding action is executed. The TCP socket on the RaspberryPi for receiving control messages is set up as shown in following Listing:

Listing 4.3: create TCP socket

```
    def create_socket(port: int):
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as
            sock:
            sock.bind(('0.0.0.0', port))
            sock.listen()
            conn, addr = sock.accept()
```

In the above listing, the option `SOCK_STREAM` defines a TCP connection. Furthermore, the option `'0.0.0.0'` is set as IP address, instructing the RaspberryPi to listen for connections on all available network interfaces (on the given port in the provided example). Hence, any ground station can simply connect to this port to send control instructions.
Some more options which are not explicitly shown are set to avoid multiple connections being established on the same port on the one hand but allow a connection to be re-established when it gets disconnected unexpectedly on the other hand.

Once a video recording has ended (either because the user manually ended the recording,

[8]The request is sent by the ground station, the drone acknowledges, and the ground station acknowledges the drone's response

or because a timeout / error occurred on the RaspberryPi's side), the temporarily saved video is transferred to the ground station. This process is implemented using TCP sockets, too. The transmission is initiated by the on-board RaspberryPi. The overall process is similar to the message sending part previously explained. Now however, the drone uses `socket.sendAll()` iteratively to send the video divided into several chunks of 1024 byte size. The ground station on the other hand uses `socket.recv(1024)` to receive the video data chunks. Once the video is completely transferred, the according file is deleted from the on-board computer and the drone is able to record a new video again.
An overview of the control flow which is used to start a measurement recording is presented in following Figure 4.19.

It can be noted that the maximal recording time is limited to 10 seconds. This ensures

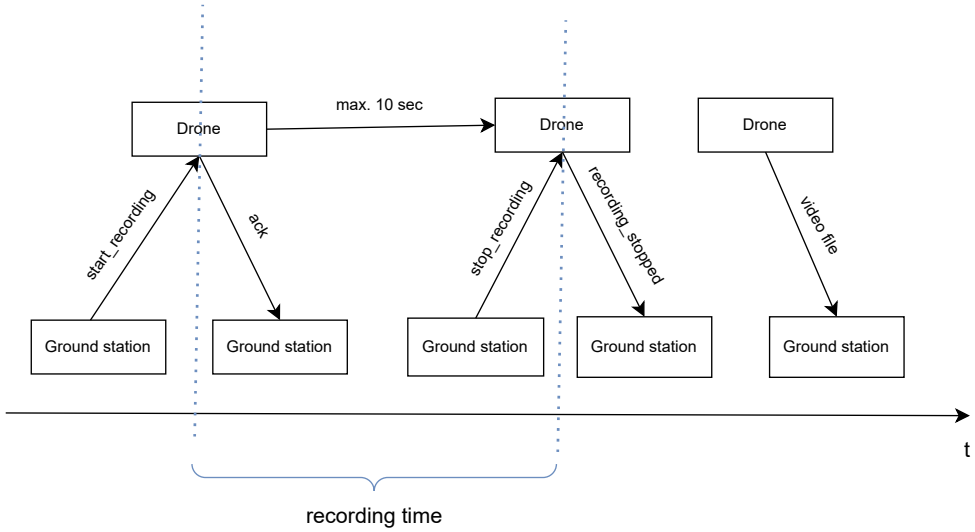


Figure 4.19.: Measurement control flow between the ground station and the drone

quick transfer times when the video is being sent to the ground station. Moreover, the implemented takeoff detection performs best on short video sequences (see subsection 4.1.4). The timeout is set to 10 seconds, as a normal long jump usually does not take longer than around seven[9] seconds.

Camera setup and video recording

Once the RaspberryPi receives the control message to start the video recording, the live-stream (see below subsubsection 4.2.3) is interrupted. Afterwards, the video recording is started. The captured video frames are saved to a temporary .mp4 file located on the on-borad RaspberryPi itself. Thus, the full available frame rate and resolution can be

[9] Approach length: 40 m, average run up speed: $6.5\frac{m}{s}$, flight time: 1 sec leads to about 7 sec total jumping time

used to record the long jump.
A frame rate of 60 FPS or higher ensures that every phase of the jump, in particular the takeoff phase, is well resolved in the time domain. The video is recorded at a resolution of 1920×1080 pixels. This resolution is sufficient for the mediapipe framework to fully recognize the athlete's posture and joints.

The camera used in this work to record long jumps is the RaspberryPi HQ camera in combination with a 16 mm C-Mount tele lens. The tele lens allows the drone to keep a safety distance to the athlete during the run-up while still keeping the athlete in focus. To manage the video recording parameters and directly interact with the camera, the Linux library `libcamera` is utilized, while the `Picamera2` Python library, built upon `libcamera`, offers a more accessible interface for this purpose.
Picamera2 especially offers the `create_video_configuration` function to conveniently set video recording parameters such as frame rate, resolution, or the option to rotate the captured frame in-place. Latter one is especially useful in this work, as the camera can thereby be mounted to the drone in any orientation while the recorded video is still orientated correctly. The video frames in this work only need to be rotated by a (multiple of) 90°, as the camera is not mounted at some arbitrary diagonal angle. Thus, the required computations can be represented by simply moving each video frame's matrix elements to a pre-defined index. A 90° rotation e.g. is performed by simply transposing the matrix (in-place) and reversing each row afterwards (applied to each RGB input channel respectively) [35].
These transformations are performed before the video frame is written to the actual video file. Hence, the post-processing overhead is significantly reduced.

Live-streaming the video feed

In addition to the video recording, a live-streaming video option is implemented to allow for aligning the drone alongside the athlete with the athlete being centered in the camera image.
The live-stream is implemented similar to the implemented communication control flow described earlier (subsubsection 4.2.3). However, package loss in this case is not as relevant as it is in the communication protocol. Hence, the live-stream is based on UDP sockets instead of TCP. sockets.
As UDP is connectionless, no overhead is introduced by connection establishment or re-transmitting lost packages. This however leads to some video frames getting (potentially) lost, which is acceptable in this case, as the video stream is not meant to be used to control the drone in real time.

To reduce the used transmission bandwidth further, the live-stream resolution is additionally limited to 640×480 pixels. The specific streaming parameters are again set using `Picamera2`'s `create_video_configuration` function as explained in the previous paragraph. To allow for higher resolution video recording, the live-stream is not available while a video recording is active.

On the ground station's side, the receiving is implemented using openCV's `VideoCapture` function which can be called with the RaspberryPi's IP address and the corresponding port number as arguments. The received video frames are then visualized using the same signal and slot mechanisms as explained in subsection 4.1.5.

4.3. Controlling the drone

Besides starting the measurement recordings and analyzing the recorded jumps, the ground station is also used to align the drone next to the athlete. Hence, a basic drone control is implemented supporting fundamental movements including forward, backward, left, right, ascending, descending, and rotation (around the drone's vertical axis).
The control in this work is based on the MAVLink protocol [36] and the corresponding python library `pymavlink` which is able to send and receive MAVLink messages (e.g. via a serial interface).
All MAVLink packets follow the same structure[10]:

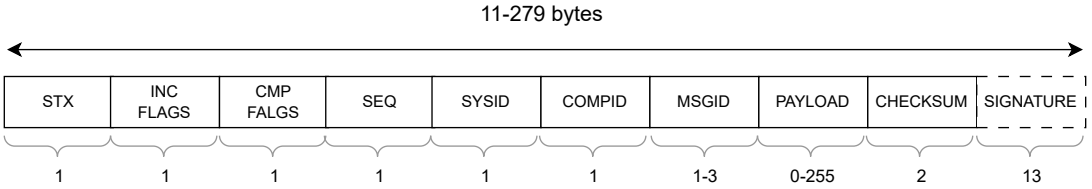


Figure 4.20.: Structure of a MAVLink message.
Below each component, their respective size in bytes is shown.
The signature is optional.
The visualization is based on the MAVLink 2.0 header shown in [36]

As this work does not use message signatures, the shown packet structure leads to a (maximal) message size of 266 bytes[11]. Due to its low overhead (11 bytes), the protocol is considered lightweight and is therefore used in many applications where low latencies are important. However, the protocol therefore does not include any security measures (apart from the message signature) like encryption.

[10] This work uses MAVLink v2. Thus, only this version's packet structure is shown.
[11] 11 bytes overhead + 255 bytes payload

In the following, each part belonging to the drone control process, namely the connection and the movement control is explained based on the above shown MAVLink message structure.

4.3.1. Connection

The control-connection between the drone and the ground station is established using two SiK telemetry radios operating at a frequency of 433 MHz with a transmission power of 100 mW. One is mounted on the drone and is directly connected to the PixHawk flight controller (see Figure 4.16), the other one is connected to the ground station using a serial connection (USB), respectively.
As MAVLink is a connectionless protocol [37], no permanent connection is established. However, as it is important to know whether the drone is still reachable by the ground station, a heartbeat signal is used to ensure a connection. Thus, the terminus *connected* will in the following be used, if a heartbeat is received from the drone. Otherwise, the drone is considered *disconnected*.
Besides receiving a heartbeat signal from the drone, the ground station emits one itself, too. By this combination of exchanging heartbeats between the two systems, a correct connection is ensured.
The camera and thus, video recording control is discussed separately in subsubsection 4.2.3 (and following).

Implementation wise, the heartbeat connection is handled in an own thread. Thus, the connection is kept alive in background while the GUI stays responsive to handle user interaction. The heartbeat is sent at a frequency of 1 Hz by the ground station and the flight controller respectively. Hence, the worst case time for detecting a connection loss lies at 1 sec. Once the heartbeat thread is started, heartbeats are sent until either the thread gets interrupted or a connection timeuout occured:

Listing 4.4: Sending mavlink heartbeats

```
    def emit_heartbeat():
        system_type = mavutil.mavlink.MAV_TYPE_GCS
        autopilot_type = mavutil.mavlink.MAV_AUTOPILOT_INVALID
        while True:
            self.connection.mav.heartbeat_send(
                system_type,
                autopilot_type,
                0,  # MAV_MODE_FLAG
                0,  # Custom mode, not used for FLYJUMP
```

```
            0,  # System status, not used for for FLYJUMP
            0,  # MAVLink version, 0: default(2.0)
        )
        time.sleep(1)
```

The provided code example uses `time.sleep(1)`. Thus, the background thread executing this function is scheduled as few as possible, ensuring that it consumes as little CPU time as possible. Moreover, the communication channel does not get overloaded. The used `heartbeat_send` function is provided by the `pymavlink` library. The system type is set to indicate the heartbeat's origin, which is the ground station in this case. The parameter `MAV_AUTOPILOT_INVALID` indicates that the heartbeat is sent by a system without any autopilot software installed.

The heartbeat receiving part is implemented using `pymavlink` , too. It is handled within the same thread as the heartbeat emitting. Hence, both functionalities are abstracted in a `DroneConnection` class which inherits from `Qhread` to make it executable in a background thread.
To receive heartbeats from the drone, the `recv_match()` function is used. This function can generally receive (wait for) every MAVLink message available. Furthermore, it can either be executed synchronously or asynchronously. The heartbeat receiving relies on the asynchronous version. If the time between two consecutive heartbeat signals exceeds two seconds, the connection is considered to be lost. The usage of asynchronous receiving ensures, that heartbeats get emitted in regular intervals and are not blocked by the receiving function.

4.3.2. Movement

Once a conrol-connection is established (see above subsection 4.3.1), the drone software supports basic movements to control the drone. All supported movement instructions are sent via the MAVLink protocol using the `pymavlink` module. Moreover, all messages are sent from a background thread. The corresponding thread is generally setup similarly to the aforementioned drone connection thread, now however, the supported movements are abstracted in one `DroneControl` class, which inherits from `QThread`. Furthermore, it needs a valid[12] `DroneConnection` object to work.

In the following the drone control is explained using multiple coordinate systems.

[12]Drone connection objects become invalid whenever the connection to the drone gets interrupted.

4.3.3. GUI control panel

Bibliography

[1] A. Seyfarth et al. "Dynamics of the Long Jump". In: *Journal of Biomechanics* 32.12 (1999), pp. 1259–1267. ISSN: 0021-9290. DOI: 10.1016/S0021-9290(99)00137-2.

[2] Murray Evans et al. "Automatic High Fidelity Foot Contact Location and Timing for Elite Sprinting". In: *Machine Vision and Applications* 32.05 (2021). DOI: 10.1007/s00138-021-01236-z.

[3] Chang Soon Tony Hii et al. "Marker Free Gait Analysis Using Pose Estimation Model". In: *2022 IEEE 20th Student Conference on Research and Development (SCOReD)*. 2022, pp. 109–113. DOI: 10.1109/SCOReD57082.2022.9974096.

[4] Valentin Bazarevsky et al. *BlazePose: On-device Real-time Body Pose Tracking*. June 17, 2020. arXiv: 2006.10204 [cs]. URL: http://arxiv.org/abs/2006.10204 (visited on February 29, 2024). preprint.

[5] Amit Gupta et al. "Knee Flexion/Extension Angle Measurement for Gait Analysis Using Machine Learning Solution "MediaPipe Pose" and Its Comparison with Kinovea®". In: *IOP Conference Series: Materials Science and Engineering* 1279.1 (March 2023), p. 012004. DOI: 10.1088/1757-899X/1279/1/012004.

[6] Google Developers. *Pose Landmark Detection Guide*. 2023. URL: https://developers.google.com/mediapipe/solutions/vision/pose_landmarker (visited on November 5, 2023).

[7] Zhe Cao et al. "Realtime Multi-Person 2D Pose Estimation Using Part Affinity Fields". In: *CoRR* abs/1611.08050 (2016). arXiv: 1611.08050. URL: http://arxiv.org/abs/1611.08050.

[8] Muhammed Kocabas et al. "MultiPoseNet: Fast Multi-Person Pose Estimation Using Pose Residual Network". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. September 2018.

[9] Daniil Osokin. "Real-Time 2D Multi-Person Pose Estimation on CPU: Lightweight OpenPose". In: *CoRR* abs/1811.12004 (2018). arXiv: 1811.12004. URL: http://arxiv.org/abs/1811.12004.

[10] *Hierarchical Data Format, Version 5*. The HDF Group. URL: https://github.com/HDFGroup/hdf5.

[11] Siu Kwan Lam et al. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. LLVM '15. New York, NY, USA: Association for Computing Machinery, November 15, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2. DOI: 10.1145/2833157.2833162.

[12] Ashish Vaswani et al. "Attention Is All You Need". In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845Abstract.html (visited on March 27, 2024).

[13] Chloe Leddy et al. "Concurrent Validity of the Human Pose Estimation Model "MediaPipe Pose" and the XSENS Inertial Measuring System for Knee Flexion and Extension Analysis During Hurling Sport Motion". In: *2023 IEEE International Workshop on Sport, Technology and Research (STAR)*. 2023 IEEE International Workshop on Sport, Technology and Research (STAR). September 2023, pp. 49–52. DOI: 10.1109/STAR58331.2023.10302442.

[14] Hao Yu et al. "Analysis and Guidance for Standing Broad Jump Based on Artificial Intelligence". In: *2023 8th IEEE International Conference on Network Intelligence and Digital Content (IC-NIDC)*. 2023 8th IEEE International Conference on Network Intelligence and Digital Content (IC-NIDC). November 2023, pp. 81–85. DOI: 10.1109/IC-NIDC59918.2023.10390658.

[15] Shuai-yu Zhou et al. "Kinematics Parameter Analysis of Long Jump Based on Human Posture Vision". In: *2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*. 2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS). July 2023, pp. 349–354. DOI: 10.1109/ISPDS58840.2023.10235733.

[16] K. Aarthy and A.Alice Nithys. "Yoga Pose Detection and Identification Using MediaPipe and OpenPose Model". In: *2023 International Conference on Computer Science and Emerging Technologies (CSET)*. 2023 International Conference on Computer Science and Emerging Technologies (CSET). October 2023, pp. 1–7. DOI: 10.1109/CSET58993.2023.10346786.

[17] Katja Ludwig et al. "Recognition of Freely Selected Keypoints on Human Limbs". In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2022, pp. 3531–3539. URL: https://openaccess.thecvf.com/content/CVPR2022W/CVSports/html/Ludwig_Recognition_of_Freely_Selected_Keypoints_on_Human_Limbs_CVPRW_2022_paper.html (visited on March 27, 2024).

[18] Katja Ludwig et al. “Detecting Arbitrary Keypoints on Limbs and Skis With Sparse Partly Correct Segmentation Masks”. In: Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision. 2023, pp. 461–470. URL: https://openaccess.thecvf.com/content/WACV2023W/CV4WS/html/Ludwig_Detecting_Arbitrary_Keypoints_on_Limbs_and_Skis_With_Sparse_Partly_WACVW_2023_paper.html (visited on March 27, 2024).

[19] Katja Ludwig et al. “All Keypoints You Need: Detecting Arbitrary Keypoints on the Body of Triple, High, and Long Jump Athletes”. In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW). June 2023, pp. 5179–5187. DOI: 10.1109/CVPRW59228.2023.00546.

[20] Klaus Mattes et al. “Kinematic Stride Characteristics of Maximal Sprint Running of Elite Sprinters – Verification of the “Swing-Pull Technique””. In: *Journal of Human Kinetics* 77 (January 30, 2021), pp. 15–24. ISSN: 1640-5544, 1899-7562. DOI: 10.2478/hukin-2021-0008.

[21] A. Seyfarth et al. “OPtimum Take-Off Techniques and Muscle Design for Long Jump”. In: *Journal of Experimental Biology* 203.4 (February 15, 2000), pp. 741–750. ISSN: 0022-0949. DOI: 10.1242/jeb.203.4.741.

[22] Kazuhiro Tsuboi. “A Mathematical Solution of the Optimum Takeoff Angle in Long Jump”. In: *Procedia Engineering*. The Engineering of Sport 8 - Engineering Emotion 2.2 (June 1, 2010), pp. 3205–3210. ISSN: 1877-7058. DOI: 10.1016/j.proeng.2010.04.133.

[23] P. Luhtanen and P. V. Komi. “Mechanical Power and Segmental Contribution to Force Impulses in Long Jump Take-Off”. In: *European Journal of Applied Physiology* 41 (1979), pp. 267–274. DOI: 10.1007/BF00429743.

[24] J. Witters et al. “A Model of the Elastic Take-off Energy in the Long Jump”. In: *Journal of Sports Sciences* (December 1, 1992). DOI: 10.1080/02640419208729949.

[25] Yuya Muraki et al. “Joint Torque and Power of the Takeoff Leg in the Long Jump”. In: *International Journal of Sport and Health Science* 6 (2008), pp. 21–32. DOI: 10.5432/ijshs.6.21.

[26] James G. Hay. “Citius, Altius, Longius (Faster, Higher, Longer): The Biomechanics of Jumping for Distance”. In: *Journal of Biomechanics*. Proceedings of the XIIIth Congress of the International Society of Biomechanics 26 (January 1, 1993), pp. 7–21. ISSN: 0021-9290. DOI: 10.1016/0021-9290(93)90076-Q.

[27] Pablo E. (Muñiz Aponte) Muñiz. “Detection of Launch Frame in Long Jump Videos Using Computer Vision and Discreet Computation”. Thesis. Massachusetts Institute of Technology, 2019. URL: https://dspace.mit.edu/handle/1721.1/123277 (visited on February 11, 2024).

[28] M. Arioli and S. Gratton. “Linear Regression Models, Least-Squares Problems, Normal Equations, and Stopping Criteria for the Conjugate Gradient Method”. In: *Computer Physics Communications* 183.11 (2012), pp. 2322–2336. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2012.05.023.

[29] Andrew Collette. *Python and HDF5: Unlocking Scientific Data*. "O'Reilly Media, Inc.", October 21, 2013. 152 pp. ISBN: 978-1-4919-4501-8. Google Books: lRCMAQAAQBAJ.

[30] P. Abascal Fuentes et al. “On the Fast Evaluation of Polynomials”. In: *Journal of Advances in Mathematics and Computer Science* (July 9, 2022), pp. 20–35. ISSN: 2456-9968. DOI: 10.9734/jamcs/2022/v37i630457.

[31] M. Tecpoyotl-Torres et al. “Real-Time Video Transmission in an FPV System Using Patch Antennas”. In: *2021 International Conference on Mechatronics, Electronics and Automotive Engineering (ICMEAE)*. 2021 International Conference on Mechatronics, Electronics and Automotive Engineering (ICMEAE). November 2021, pp. 251–256. DOI: 10.1109/ICMEAE55138.2021.00047.

[32] Matko Silic et al. “QoE Assessment of FPV Drone Control in a Cloud Gaming Based Simulation”. In: *2021 13th International Conference on Quality of Multimedia Experience (QoMEX)*. 2021 13th International Conference on Quality of Multimedia Experience (QoMEX). June 2021, pp. 175–180. DOI: 10.1109/QoMEX51781.2021.9465385.

[33] V. Cerf and R. Kahn. “A Protocol for Packet Network Intercommunication”. In: *IEEE Transactions on Communications* 22.5 (May 1974), pp. 637–648. ISSN: 1558-0857. DOI: 10.1109/TCOM.1974.1092259.

[34] *Specification of Internet Transmission Control Program*. Request for Comments RFC 675. Internet Engineering Task Force, December 1974. 70 pp. DOI: 10.17487/RFC0675.

[35] Paul Godard et al. “Efficient Out-of-Core and Out-of-Place Rectangular Matrix Transposition and Rotation”. In: *IEEE Transactions on Computers* 70.11 (November 2021), pp. 1942–1948. ISSN: 1557-9956. DOI: 10.1109/TC.2020.3030592.

[36] Anis Koubâa et al. “Micro Air Vehicle Link (MAVlink) in a Nutshell: A Survey”. In: *IEEE Access* 7 (2019), pp. 87658–87680. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2924410.

[37] Sukhrob Atoev et al. “Data Analysis of the MAVLink Communication Protocol”. In: *2017 International Conference on Information Science and Communications Technologies (ICISCT)*. 2017 International Conference on Information Science and Communications Technologies (ICISCT). November 2017, pp. 1–3. DOI: 10.1109/ICISCT.2017.8188563.

Appendix

A. Drone

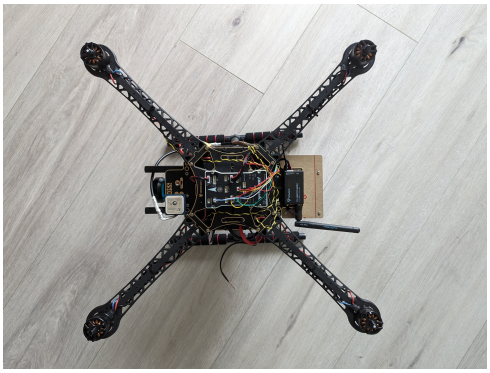


Figure 1.: Fully equipped drone - Top view.
The PixHawk flight controller is on top of the drone.

Figure 2.: Drone with a RaspberryPi camera mounted.