POLITECNICO DI TORINO

DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATIONS

Master's degree course in Mechatronic Engineering

Develop a self-sufficient system capable of detecting people's physiological parameters during Search and Rescue (SAR) missions

Advisor

Prof. Massimo VIOLANTE

Co-advisors

Dr. Jacopo SINI

Dr. Luigi PUGLIESE

Company Reference

Gea CARENA, M.Eng

Candidate

Nicolò FANTASIA

Academic Year
2023/2024

Abstract

The main objective of this thesis is to develop a self-sufficient system capable of detecting people's physiological parameters during Search and Rescue (SAR) missions. Specifically, we focused on creating a battery-powered device capable of counting the number of people in an image and extracting their physiological parameters such as Heart and Respiration rate using the remote photoplethysmography (RPPG) technique. The goal is to integrate this system into a rover or a drone, allowing an operator to control it remotely. In the first part of the study, we focused on hardware selection. It was necessary to identify a sufficiently powerful board capable of handling the computational load of the program while being lightweight, compact, and with low power to allow it to operate on battery. For this reason, we chose the Raspberry Pi 5. After carefully selecting all device components, with particular attention to the cameras, an in-depth analysis was current requirements of each component, we determined the need for a 10,000 mAh power bank capable of delivering 5V/5A from a single port. However, since no commercially available power banks met these specifications, we opted for a model with the required capacity but with a maximum output of 12V/3A from a single port. We adjusted the power supply using a trigger decoy board and a step-down converter to obtain the necessary input values for the system. In the second part of the study, we focused on developing an application, based on open-source frameworks, to implement the aforementioned functionalities. For people counting, we used YOLO (You Only Look Once), a computer vision algorithm for real time body and object detection. For remote photoplethysmography, the script was developed using MediaPipe and HeartPy. MediaPipe allowed us to extract, through FaceMesh, the forehead and cheek regions of the face, which serve as input for HeartPy, an open source library for heart rate analysis. Finally, during the testing phase, we verified that, under standard conditions (direct natural light and both direct and diffuse artificial light), the obtained results were promising, with heart rate estimation errors below 10 bpm. The main goal of this project is to enable an operator to view in real-time the images captured by the device, so potential future developments include integrating a stabilized camera (particularly useful for drone applications), implementing motors to allow the operator to adjust the camera's orientation, and using an infrared light source with a dedicated camera to improve performance in low-light conditions.

Dedica

Summary

List of Tables

List of Figures

Chapter 1

The choice of components

1.1 Introduction

The purpose of this study is to design and develop a system, both at the hardware and software levels, that can be integrated into a drone or a rover. This system aims to assist rescue teams in identifying the number of individuals in need of assistance and providing critical information about the health status of each person. By combining advanced hardware and software capabilities, the proposed solution seeks to enhance the efficiency and effectiveness of search and rescue missions, particularly in challenging and time-sensitive environments.

For this reason, the selection of hardware is arguably the most critical aspect of this study. The entire suite of dedicated software must be able to operate seamlessly and without real-time latency. The hardware components must therefore provide sufficient computational power and reliability to support the demanding requirements of this application, ensuring the smooth operation of algorithms for image processing, facial recognition, and health monitoring.

In this chapter, we will delve into the selection of hardware components, thoroughly analyzing the reasoning and considerations behind

each choice. The discussion will highlight the factors that influenced the decisions, ensuring that the chosen hardware aligns with the system's objectives and operational requirements.

1.2 Hardware Selection

The selection of hardware was primarily driven by the goal of finding a machine that is not only as powerful as possible, but also capable of supporting high-quality image acquisition peripherals. Additionally, it was essential to choose a system that is compact in size and lightweight, ensuring ease of integration into the design while maintaining portability and efficiency. In this context, the hardware needed to strike a delicate balance between computational power, imaging capabilities, and physical constraints. The system had to be robust enough to handle demanding tasks, such as real-time image processing and facial recognition, while also being small enough to fit seamlessly into a drone or rover platform, ensuring mobility and ease of use in field conditions. The final choice of hardware had to meet these multifaceted requirements, ensuring that the system could operate efficiently without compromising on performance or practicality.

1.2.1 Single Board Computer

The decision to use the Raspberry Pi 5 with 8GB of RAM for this project was based on several compelling reasons, making it an optimal choice for our system. The Raspberry Pi 5, with its powerful hardware specifications, offers a robust platform capable of handling demanding tasks such as real-time image processing and facial recognition. The 8GB of LPDDR4-3200 RAM ensures smooth performance even when running memory-intensive applications, which is crucial for tasks that involve processing large amounts of data, such as high-resolution image

acquisition and analysis.

Physically, the Raspberry Pi 5 is a compact single-board computer, measuring just 90mm x 60mm, with a weight of approximately 46 grams. Despite its small size, it offers impressive computational power thanks to its quad-core ARM Cortex-A76 processor running at 2.0 GHz. This combination of size, power, and efficiency makes the Raspberry Pi 5 the perfect candidate for use in portable systems like drones or rovers, where space and weight constraints are critical.

In addition to its processing power, the Raspberry Pi 5 is equipped with a wide range of connectivity options, including USB 3.0 and USB 2.0 ports, Gigabit Ethernet, Wi-Fi 6, and dual micro-HDMI ports supporting 4K output. The board also includes a camera interface with two MIPI CSI connectors, making it ideal for connecting high-quality camera modules for image acquisition.

Another significant advantage of the Raspberry Pi 5 is its power supply. The device is powered via a USB-C port, which supports Power Delivery (PD) technology. This allows us to power the Raspberry Pi through a carefully modified power bank, which is particularly useful for portable applications like drones or rovers. The ability to use a power bank with USB-C PD support ensures a reliable, long-lasting power source, essential for operating the Raspberry Pi 5 in remote or field environments without the need for traditional power outlets.

Raspberry Pi 5 specifications[1]:

Figure 1.1: Image of a raspberry pi 5

- Processor: Quad-core ARM Cortex-A76, 2.0 GHz
- RAM: 8 GB LPDDR4-3200
- Storage: MicroSD card slot (supports up to 1TB)
- USB Ports: 2x USB 3.0, 2x USB 2.0
- Networking: Gigabit Ethernet, Wi-Fi 6 (802.11ax)
- Video Output: 2x micro-HDMI (supports up to 4K resolution)
- Camera Interface: 2x MIPI CSI connectors for high-quality camera modules
- GPIO: 40-pin header, compatible with various sensors and peripherals
- Power Supply: USB-C with Power Delivery (PD) support for efficient and portable power solutions

The Raspberry Pi 5 with 8GB of RAM not only meets the technical requirements for image processing but also provides the flexibility and expandability needed to integrate with a wide range of sensors and devices, ensuring the system can evolve to meet future demands. Its compact design, coupled with its powerful processing capabilities

and efficient power delivery system, makes it a perfect fit for portable applications in challenging environments, such as search and rescue missions.

Given the significant computational load and the intention to overclock the machine, the need to cool the board and maintain it at a constant operating temperature, below the thermal throttling threshold, cannot be overlooked. It was therefore decided to use a device from Raspberry Pi, namely the Raspberry Pi Active Cooler, which, through a passive heatsink and a fan, should keep the board's temperatures under control. This device is applied directly to the card in contact with the CPU and GPU via thermal pads and is powered by the card itself.

Figure 1.2: Our raspberry Pi 5 in its case with the Active Cooler applied

1.2.2 Battery Module

It is crucial that the system operates independently from the main electrical grid, as it is intended to be deployed in remote or inaccessible areas. For this reason, the entire system is designed to be powered by a battery. To ensure proper functionality and sufficient autonomy, the current required by each component was calculated using the formula:

$$I = \frac{P}{V}$$

where I is the current in milliamperes (mA), P is the power consumption in watts (W), and V is the operating voltage in volts (V).

The table below summarizes the power requirements of the primary components of the system:

Table 1.1: Power Requirements of System Components[1] [2][3] [4][5]

Component	**[W]**	**[V]**	**[mA]**
Raspberry Pi 5 8GB RAM	10.0	5	2000
Raspberry Pi 5 Active Cooler	1.0	5	200
Raspberry camera module v3 (autofocus)	1.5	5	300
Raspberry camera module v3 Wide (autofocus)	1.5	5	300
Raspberry HQ camera C-CS Mount (manual focus)	2.0	5	400
Radar Doppler	2.5	5	500

The Raspberry Pi 5 will be connected via a USB-C to USB-C cable to a decoy trigger equipped with Power Delivery (PD) technology. This decoy requests the power bank to supply 12V at 3A. The power is then delivered to a step-down converter through properly dimensioned cables. The step-down converter regulates the output to 5V at a maximum of 5A, which is provided to the Raspberry Pi through a USB-A to USB-C cable. This configuration ensures stable and efficient power delivery to the board and all connected peripherals.

The choice of cables, especially the USB cables, is a critical aspect of the system's design. USB-C cables with a power rating of up to 100W and support for Power Delivery (PD) technology were selected to ensure both compatibility and reliability. These cables are essential for negotiating the correct power profile with the battery module, allowing it to deliver the necessary voltage and current without interruption. Furthermore, these high-capacity cables ensure that the 5A current

required by the Raspberry Pi 5 and its peripherals can be transported safely and efficiently, avoiding voltage drops or overheating issues.

The combination of a robust decoy trigger, properly rated cables, and a high-efficiency step-down converter ensures that the Raspberry Pi 5 and its connected components receive stable power, maintaining reliable operation in all scenarios. This meticulous power management setup is critical for achieving the autonomous functionality required by the system.

Powerbank

As mentioned earlier, the choice of the powerbank was primarily driven by the need to have a device capable of delivering $12V/3A$ from a single output port. This requirement is essential for ensuring stable and sufficient power supply for the Raspberry Pi 5 and its associated components during operation.

For the prototype, we plan to utilize only one camera module, simplifying the power demand calculation. From the total current values listed in Table 1.1, the estimated power consumption of the system amounts to:

$$I_{\text{estimated}} = 3460\,\text{mA}$$

To ensure the system is resilient to factors such as power surges, cable resistance losses, and potential overheating, I introduced a safety factor of 1.2 in the current calculation. This accounts for fluctuations and unexpected power draw. The total current requirement, including this safety margin, is calculated as:

$$I_{\text{required}} = \frac{3460}{1000} \times 1.2 = 4.152\,\text{A}$$

Regarding the dimensioning of the powerbank itself, I considered

a hypothetical usage time of 2.5 hours, which is significantly longer than what I expect for the actual operational timeframe. To estimate the energy requirement, I multiplied the current consumption by the usage time and an additional safety coefficient of 1.15, which accounts for potential underperformance of the powerbank's actual capacity compared to its nominal rating. The required capacity is therefore calculated as:

$$\text{Capacity}_{\text{required}} = 3460 \times 2.5 \times 1.15 = 9947.5\,\text{mAh}$$

Based on this calculation, I selected a powerbank with a nominal capacity of 10,000 mAh. This ensures sufficient power availability for the system's operation, even accounting for inefficiencies or unexpected power draw during use.

Figure 1.3: Image of the chosen powebank

Decoy Trigger

In order to trigger the powerbank and make it supply 12V/3A, a decoy trigger board will be used. The selection of this decoy was based on specific requirements, as the board must be capable of handling 12V input, feature a USB-C port, and support the USB Power Delivery (PD) 2.0 standard. The chosen board meets these criteria, as it is equipped with a USB-C input port and DIP switches. These DIP switches allow for voltage adjustments, enabling the board to handle a range of voltages from 5V to 20V, depending on the position of the switches. Therefore, this decoy will act as a bridge between the powerbank and the step-down converter, ensuring that the correct voltage is supplied to the system.

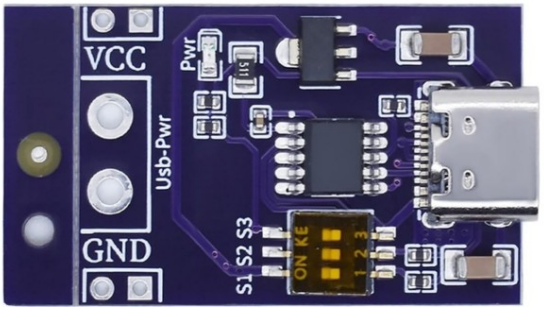


Figure 1.4: Image of the choosed Decoy

Step Down DC/DC Converter

The step-down converter is a crucial component in the system, as it is used to reduce the input voltage from 12V to 5V, ensuring compatibility with the Raspberry Pi. This converter receives its input voltage directly from the decoy trigger board, which manages the power output of the powerbank. The step-down board provides its output through a USB-A port, delivering a stable 5V supply.

To connect the step-down converter to the Raspberry Pi, a high-quality USB-A to USB-C cable is used. This ensures efficient power transfer to the Raspberry Pi while maintaining a reliable connection. The use of this step-down converter allows for safe and stable operation of the Raspberry Pi, which requires a consistent 5V power input for its functionality.

Figure 1.5: Image of the choosed step down

1.2.3 Cameras

Camera Selection

The choice of the camera is crucial for the intended application because obtaining a sharp and high-resolution image is essential. The entire software pipeline heavily depends on the quality of the input image, in addition to the computational power of the Raspberry Pi board. For our system, the image of the subject's face must be at least 100-150 pixels wide to be effectively processed by OpenCV, ensuring acceptable accuracy in measurements and estimations derived from the software.

Several key parameters were considered during the camera selection process:

- **F-stop (Aperture)**: To control the light entering the camera and depth of field.
- **Sensor Diagonal**: The physical size of the sensor affecting field of view (FOV) and resolution.

- **Focal Length**: To determine the magnification and FOV.
- **Field of View (FOV)**: Measured in degrees, essential for capturing the required scene.
- **Sensor Resolution**: Specified as width × height in pixels, determining the level of detail in the captured image.

Using these parameters, we calculated the **circle of confusion (c)**, which is an important factor in determining depth of field. The formula used is:

$$c = \frac{\text{Diagonal of the sensor}}{1500}$$

Subsequently, the **Depth of Field (DOF)** was calculated. DOF is the range within which objects appear acceptably sharp in an image and is divided into one-third in front of the subject and two-thirds behind. The DOF is determined using the following formula:

$$DOF = \frac{2 \cdot N \cdot c \cdot u^2 \cdot f^2}{f^4 - N^2 \cdot c^2 \cdot (u - f)^2}$$

Where:

- N: F-number (focal ratio or aperture setting).
- c: Circle of confusion.
- u: Distance to the subject.
- f: Focal length.

These calculations have been instrumental in selecting a camera that meets the system's requirements for accuracy and reliability.

After calculating the sensor dimensions in terms of width and height, and considering the average dimensions of a human face (both height and width), we proceeded to calculate the Field of View (FOV) for both width and height at varying distances ranging from 1 meter to 5 meters. These calculations were performed using the following formulas:

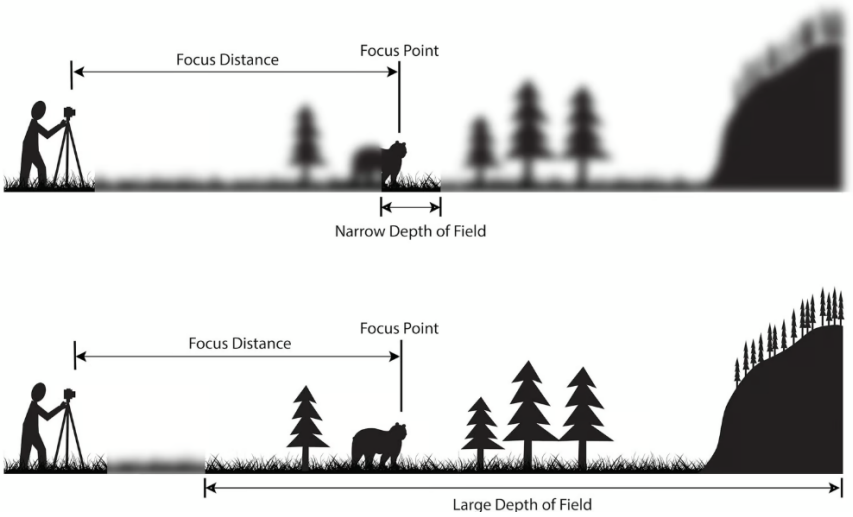


Figure 1.6: Image of a general DOF

$$FOV_w = \frac{S_w \cdot D}{f}, \quad FOV_h = \frac{S_h \cdot D}{f}$$

Where:

- FOV_w: Field of View in width.
- FOV_h: Field of View in height.
- S_w: Sensor width.
- S_h: Sensor height.
- D: Distance to the subject.
- f: Focal length of the lens.

Using the calculated FOV values, we then determined the dimensions of the face in pixels. This was achieved by considering the real-world dimensions of the face (O_w for width and O_h for height), the resolution of the sensor (R_w for width and R_h for height), and the FOV. The pixel dimensions were computed using the following formulas:

$$O_{pixel\_w} = \frac{O_w \cdot R_w}{FOV_w}, \quad O_{pixel\_h} = \frac{O_h \cdot R_h}{FOV_h}$$

Where:

- O_{pixel_w}: Face width in pixels.
- O_{pixel_h}: Face height in pixels.
- O_w: Real-world face width.
- O_h: Real-world face height.
- R_w: Sensor resolution in width (in pixels).
- R_h: Sensor resolution in height (in pixels).
- FOV_w: Field of View in width (in real-world units).
- FOV_h: Field of View in height (in real-world units).

These calculations allowed us to verify whether the face dimensions in pixels at various distances meet the required threshold of 100-150 pixels in width. This is essential to ensure that the captured image provides sufficient resolution for OpenCV to process and for the software to make accurate measurements and predictions.

We opted to use Raspberry Pi cameras for this project. The primary motivation for this choice is the seamless hardware and software compatibility these cameras offer with the Raspberry Pi boards. I include below the calculations and procedures just explained, along with the numerical results that guided us in selecting the cameras.

Raspberry camera module v3

The Raspberry Camera Module v3 was the first camera considered for our prototype, primarily due to its autofocus capability. This feature

is particularly advantageous in challenging conditions, such as when the camera is mounted on a drone where vibrations and movement can affect image clarity. The autofocus ensures sharper images, which is critical for accurate processing.
The specifications of the Raspberry Camera Module v3 are as follows[2]:

Parameter	**Value**	**Unit**
N (f-stop)	1.8	-
d (diagonal sensor)	7.4	mm
u (subject distance)	2000	mm
f (focal length)	4.74	mm
$\mathbf{FOV_l}$	66	degrees
$\mathbf{FOV_h}$	41	degrees

Table 1.2: Key Parameters of the Raspberry Camera Module v3.

The circle of confusion (c) was calculated using the formula:

$$c = \frac{\text{Diagonal of the sensor}}{1500} = 7{,}4/1500 = 0.005$$

The depth of field (DOF) was calculated as follows:

$$\text{DOF} = 8362.28\,\text{mm}$$

The physical dimensions and resolution of the sensor are presented in the table below:

Parameter	**Value**	**Unit**
Sw (sensor width)	6.45	mm
Sh (sensor height)	3.63	mm
Rw (sensor resolution width)	4608	px
Rh (sensor resolution height)	2592	px
Ow (face width)	150	mm
Oh (face height)	200	mm

Table 1.3: Sensor Dimensions and Resolution.

We conducted tests to calculate the Field of View (FOV) and the pixel dimensions of a face at various distances. The FOV calculations

and the pixel dimension of the faces are computed with the formulas indicated before.

The results of these calculations are presented in the table below, where the column $\mathbf{O_{pixel_w}}$ has been highlighted in green:

Test	Distance	$\mathbf{FOV_w}$	$\mathbf{FOV_h}$	$\mathbf{O_{pixel_w}}$	$\mathbf{O_{pixel_h}}$
1m	1000 mm	1360.76	765.82	507.95	676.92
2m	2000 mm	2721.52	1531.65	253.98	338.46
3m	3000 mm	4082.28	2297.47	169.32	225.64
4m	4000 mm	5443.04	3063.29	126.99	169.23
5m	5000 mm	6803.80	3829.11	101.59	135.38

Table 1.4: Field Test Results.

Figure 1.7: Image of the Raspberry camera module v3

Raspberry camera module v3 wide

The Raspberry Camera Module v3 is the same sensor as the previous one, with the key difference being that it has a larger field of view (FOV). This increased FOV could be particularly useful when using a rover, as it allows for a wider image capture, which is advantageous for better situational awareness and navigation. Additionally, the autofocus capability of the camera ensures sharper images in dynamic

and challenging environments, such as when the camera is mounted on a rover or drone, where vibrations could affect the image quality.

The specifications of the Raspberry Camera Module v3 are as follows[3]:

Parameter	**Value**	**Unit**
N (f-stop)	2.2	-
d (diagonal sensor)	7.4	mm
u (subject distance)	500	mm
f (focal length)	2.75	mm
$\mathbf{FOV_l}$	102	degrees
$\mathbf{FOV_h}$	67	degrees

Table 1.5: Key Parameters of the Raspberry Camera Module v3.

The circle of confusion (c) was calculated using the formula:

$$c = \frac{\text{Diagonal of the sensor}}{1500} = 7{,}4/1500 = 0.005$$

The depth of field (DOF) was calculated as follows:

$$\text{DOF} = 1462.25\,\text{mm}$$

The physical dimensions and resolution of the sensor are presented in the table below:

Parameter	**Value**	**Unit**
Sw (sensor width)	6.45	mm
Sh (sensor height)	3.63	mm
Rw (sensor resolution width)	4608	px
Rh (sensor resolution height)	2592	px
Ow (face width)	150	mm
Oh (face height)	200	mm

Table 1.6: Sensor Dimensions and Resolution.

We conducted tests to calculate the Field of View (FOV) and the pixel dimensions of a face at various distances. The FOV calculations and the pixel dimension of the faces are computed with the formulas indicated before.

The results of these calculations are presented in the table below, where the column $\mathbf{O_{pixel_w}}$ has been highlighted in green:

Test	**Distance**	$\mathbf{FOV_w}$	$\mathbf{FOV_h}$	$\mathbf{O_{pixel_w}}$	$\mathbf{O_{pixel_h}}$
1m	1000 mm	2345.45	1320	294.70	392.73
2m	2000 mm	4690.91	2640	147.35	196.36
3m	3000 mm	7036.36	3960	98.23	130.91
4m	4000 mm	9381.82	5280	73.67	98.18
5m	5000 mm	11727.27	6600	58.94	78.55

Table 1.7: Field Test Results.

Figure 1.8: Image of the Raspberry Camera Module v3 wide

As seen from the table, at distances of 4 and 5 meters, this camera does not reach the threshold value of 150 pixels. As a result, its use is limited to a fairly short range

Raspberry HQ camera with 8mm CS-Mount with adjustable aperture

This time, we are using a high-quality Raspberry Camera Module sensor, which, unlike previous models, does not feature autofocus. Instead, it utilizes a C/CS mount for attaching lenses, offering greater flexibility and complexity. In this case, we are using an 8mm lens with an adjustable aperture, which provides more control over image sharpness and depth of field. However, it requires manual adjustment of the lens for optimal performance.

The specifications of the Raspberry Camera Module v3 with the new lens are as follows[4][6]:

Parameter	**Value**	**Unit**
N (f-stop)	1.2	-
d (diagonal sensor)	7.6	mm
c (circle of confusion)	0.005066667	mm
u (subject distance)	3000	mm
f (focal length)	8	mm
$\mathbf{FOV_l}$	58.4	degrees
$\mathbf{FOV_h}$	44.6	degrees

Table 1.8: Key Parameters of the Raspberry Camera Module v3 with 8mm lens.

The circle of confusion (c) was calculated using the formula:

$$c = \frac{\text{Diagonal of the sensor}}{1500} = \frac{7.6}{1500} = 0.005066667\,\text{mm}$$

The depth of field (DOF) was calculated as follows:

$$\text{DOF} = 1860.29789\,\text{mm}$$

The physical dimensions and resolution of the sensor are presented in the table below:

We conducted tests to calculate the Field of View (FOV) and the pixel dimensions of a face at various distances. The FOV calculations

Parameter	Value	Unit
Sw (sensor width)	6.287	mm
Sh (sensor height)	4.712	mm
Rw (sensor resolution width)	4056	px
Rh (sensor resolution height)	3040	px
Ow (face width)	150	mm
Oh (face height)	200	mm

Table 1.9: Sensor Dimensions and Resolution.

and the pixel dimensions of the faces are computed with the formulas indicated before. The results of these calculations are presented in the table below, where the column $\mathbf{O_{pixel_w}}$ has been highlighted in green:

Test	Distance	$\mathbf{FOV_w}$	$\mathbf{FOV_h}$	$\mathbf{O_{pixel_w}}$	$\mathbf{O_{pixel_h}}$
1m	1000 mm	785.88	589	774.17	1032.26
2m	2000 mm	1571.75	1178	387.08	516.13
3m	3000 mm	2357.63	1767	258.06	344.09
4m	4000 mm	3143.50	2356	193.54	258.06
5m	5000 mm	3929.38	2945	154.83	206.45

Table 1.10: Field Test Results.

Figure 1.9: Images of the Raspberry High Quality Camera and the choosen 8mm lens

1.3 3D Printed Support

For our application, a stable support for the camera is essential. Therefore, together with Marco Chabod from R3DSIGN, we developed a support and two different cases for the cameras: one for the HQ camera and one for the v3 and v3 wide cameras, which share the same shape

The support is a plate with holes, rounded corners, and grooves for M4 nuts, with a raised neck and a fork mount with grooves and a transverse hole. The plate has been designed to follow the curvature of the Raspberry Pi 5 case, and the holes for the nuts are positioned in a way that does not obstruct the board's operation. Furthermore, the off-axis position of the neck was designed to avoid bending the flat cable more than necessary, as these are very delicate components.

The support was then fixed with 3 M4 bolts to the top of the Raspberry Pi case, which was drilled using a column drill.

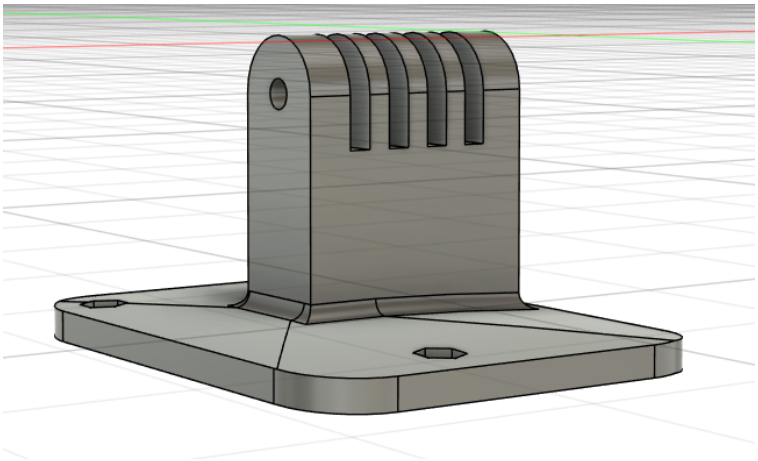


Figure 1.10: Design of the support

The two cases were designed to protect the camera PCBs without covering the flat cable connection, making the mounting and dismounting of the cameras safer and quicker. Both supports have a fork mount with grooves and a through hole aligned with the one in the support, allowing a through bolt to secure the camera to the support.

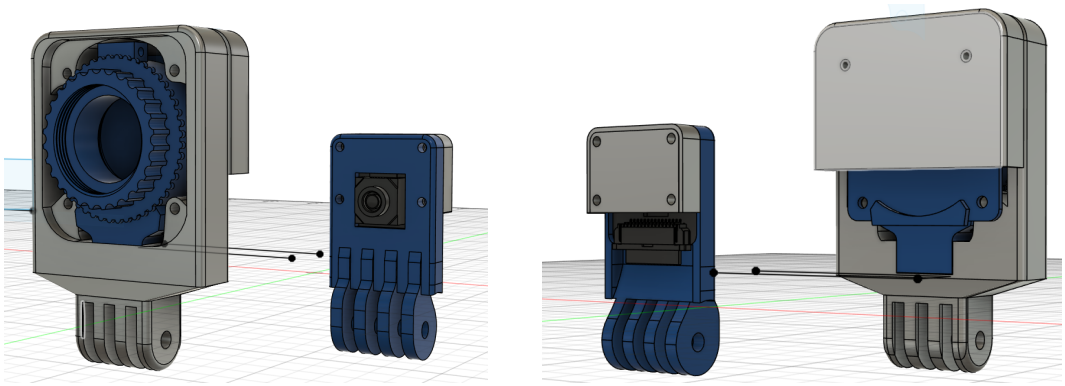


Figure 1.11: Image of the cases of the HQ camera and the v3s

The supports and cases were printed using a **Bambu Lab P1S**, with PLA as the material, particularly for its low printing temperature, making it ideal for prototyping.

Figure 1.12: The raspberry Pi5 with all the 3D printed parts

Chapter 2

Code description

2.1 Introduction

In this chapter, we will discuss the functionality of the software. The main goal of the code is to monitor heart rate, age, and gender, as well as perform real-time people counting. The software is written in Python and developed using Visual Studio Code as the IDE. Its core is based on OpenCV (Open Source Computer Vision Library), an open-source library that includes hundreds of algorithms for computer vision[7] that we used to have acces to the webcam image and make possible all the necessary analysis and predictions.
Morover, we used a virtual environment to isolate the necessary dependencies and packages, thus avoiding conflicts between libraries. For this reason, Conda, specifically its more complete distribution, Anaconda, was chosen as the package manager.[8]

In addition to OpenCV, other frameworks and libraries have been used, including Mediapipe and HeartPy for heart rate calculation, and YOLO for body pose estimation and people counting. Mediapipe is a framework developed by Google that provides tools and libraries for multimedia image processing in real time, based on machine learning and computer vision technologies [9] [10]. HeartPy, on the other hand, is an open-source toolkit developed in Python for heart rate analysis using

photoplethysmogram (PPG) technology[11]. By accurately extracting portions of the face through Mediapipe, it is possible to calculate the subject's heart rate with a certain margin of error. Regarding people counting, the choice was made to use YOLO (You Only Look Once), a real-time object detection algorithm widely used in the field of computer vision.[12] .

The software offers several modes of use: people counting only, heart rate/age/gender recognition only, or both functionalities active simultaneously. To manage these modes, multithreading technology was implemented. However, the version of Python used on the development machine (Python 3.10.11), which is the last one compatible with Mediapipe on Windows environments[13], still uses the Global Interpreter Lock (GIL). This mechanism enforces that only one thread at a time can execute Python code, preventing true multithreading[14]. To work around this problem, Python manages execution by "switching" between different threads.

Events and queues were also used to allow threads to share information and trigger events only when necessary, and to improve readability, many of the functions used in the code have been organized into separate files.

In the following sections, we will analyze the main structure of the code and the code blocks where measurements and predictions are made.

2.2 Main structure of the code

In this section, we will focus on how the code actually works, providing an overview of its execution before diving into the individual blocks of code in the following sections.

The first part concerns the imports, where we include all the modules

and packages required for the execution of the code, which are available in the `requirements.txt` file and can be installed using a simple `pip` command.

For convenience, I divided the imports based on how the software uses them, specifically:

```
#GENERAL IMPORTS
import cv2
from threading import Thread, Event

#HR RECOGNITION IMPORTS
import heartpy as hp
from queue import Queue
from datetime import datetime
import mediapipe as mp
import csv
import numpy as np

#BODYPOSE/PEOPLE COUNTING IMPORTS
from ultralytics import YOLO
```

The GENERAL IMPORTS refer to the main structure of the code, meaning the use of OpenCV, the threading and event system.

The **HR RECOGNITION IMPORTS** refer to the modules used for heart rate prediction. Specifically, the following are imported:

- `mediapipe` for creating the face mesh used to extract portions of the face for heart rate prediction.
- `heartpy` and `numpy` for the actual heart rate prediction.
- `queue`, `datetime`, and `csv` for creating shared queues between threads and constructing and filling the CSV files where measurements are saved.

Regarding the **BODY POSE/PEOPLE COUNTING IMPORTS**, the `YOLO` module is imported to load and use the selected model.

Next, the output files for the calculated values, the queues that are needed for sharing data between threads, and the event used to trigger

heart rate calculations are defined:

```
output_file_1 = "heartpy_results_total.csv"
output_file_2 = "heartpy_results_forehead.csv"
output_file_3 = "heartpy_results_left.csv"
output_file_4 = "heartpy_results_right.csv"

data_queue = Queue()
bpm_queue = Queue()
time_queue = Queue()
hr_analysis_event = Event()
```

Then, the files containing the additional functions are imported, organized according to the main topics to make the file more readable and organized, followed by the setup for the models used to generate predictions.

```
#FUNCTION IMPORTS
import General_utils as utils
import Face_detection_utils as fd
import Body_pose_utils as bp  # currently empty
import Age_gender_utils as ag
import HeartRate_Functions as hr

#SET UP MEDIAPIPE
mp_drawing = mp.solutions.drawing_utils  # type: ignore
mp_face_mesh = mp.solutions.face_mesh  # type: ignore

#LOAD AGE GENDER MODELS
age, gender = ag.ag_model_loads()

#LOAD YOLO MODEL
model = YOLO('Models/Yolo/yolo11n.pt')
```

In particular, the age and gender prediction model is loaded using a function defined in the external `Age_Gender_utils` file.

The function `ag_model_loads()` defines the paths for the `.prototxt` and `.caffemodel` files, which contain the configuration of the neural network and the weights trained by the model. Then, the function `cv2.dnn.readNet()` is used to load the neural networks using the mentioned configuration files.

In the final part of the code initialization, OpenCV is set up to access the camera feed (webcam), adjust the auto-exposure settings (if supported by the device), and select the frame size to be analyzed, which depends on the device.

We can now proceed to the actual functioning of the code. As mentioned earlier, it is divided into two threads called `main` and `analyze_heartbeat`. When the program starts, it enters the `main` thread, setting the `test` variable to its default value, which represents one of the three modes of code execution:

- "0" People Counting
- "1" HR, Gender/Age recognition
- "2" Both

The default mode at startup is, in this case, "People Counting" (`Test=0`).

The `video_writer = None` variable is then defined, which will be used shortly to start recording video of the test images.

Next, the mediapipe model is loaded with a `with` statement, and after generating the file and writing the header with the `write_header_file` function (if the file is not already present) called from the "general_utils" file, and declaring the variables that will hold the values of the shared queue between threads, the first operations on the image are executed.

There is a check to see if the camera feed is received correctly.If the test is successful, the image is:

1. Converted from RGB to BGR color space
2. Flipped
3. Processed by Mediapipe for the computation of the face mesh
4. Converted back to RGB

5. Overlayed to show graphically MediaPipe's execution results

The BGR conversion is crucial because it allows mediapipe to perform face mesh detection[15]. Then, the FPS of the feed is calculated, and the image dimensions are obtained so that the FPS value can be displayed on the image.

The code checks if the `video_writer` is None and initializes the video output file, the pixel format, and the codec to use, which in this case is H264 obtaining a .mp4 file. It is important to resize all the frames to the same standard resolution (1280x768, 1820x1080) to obtain a valid video output since otherwise we will obtain an empty file without any exception raised by openCV.

The `main` thread contains two important blocks of code: the first performs the heart rate calculation and the age/gender prediction, and the second handles body pose and people counting. These two blocks will be referred to as **BLOCK HR 1** and **BODY POSE/PEOPLE COUNTING BLOCK** for simplicity, and they will be explained in the next paragraphs. It is important to understand how the flow works in the modes.

Before the **HR BLOCK 1**, there is an `if` statement that, when the `Test` variable is set to 1 or 2 (HR/Gender/Age recognition mode, or Both mode), executes this part of the code, which, along with the `analyze_heartbeat` thread containing **HR BLOCK 2**, calculates the heart rate and displays the gender and age predictions.

Similarly, there is another `if` statement that checks if the `Test` variable is set to 0 or 2 (People counting mode or Both mode), leading to the execution of the portion of the code that we referred at with **PEOPLE COUNTING BLOCK**.

At this point, a function from the `general_utils` file is called to display the execution mode (depending on the `Test` variable) on the image, resize the image, display the real-time video, and start recording it.

Figure 2.1: Text on the video that shows the buttons for changing the execution mode and the current mode

Here is a small extract of the `main` thread:

```
def main(cap, data_queue, bpm_queue, time_queue):
    # Test Variables
    # 0 People Counting
    # 1 HR, Gender/Age recognition
    # 2 Both
    Test = 0  # Default value People Counting
    Stream=0 # default stream value

    video_writer = None

    with mp_face_mesh.FaceMesh(static_image_mode=True,
   max_num_faces=6, min_detection_confidence=0.5) as facemesh:

        while cap.isOpened():
            ret, frame = cap.read()

            # Pre-operation with the image
            image = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            image = cv2.flip(image, 1)

            results = facemesh.process(image)
            image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)

            fps = cap.get(cv2.CAP_PROP_FPS)
            frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
            frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

            cv2.putText(image, f"FPS:{fps}", (frame_width - 110,
   frame_height - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255),
    1)

            if video_writer is None:
                fourcc = cv2.VideoWriter_fourcc(*'MP4V')

                video_writer = cv2.VideoWriter("test_1.avi",
   fourcc, fps, (912, 480))
```

```
33
34              if Test == 1 or Test == 2:
35                  # HR BLOCK 2
36                  pass
37
38              if Test == 0 or Test == 2:
39                  # BODY POSE/PEOPLE COUNTING BLOCK
40                  pass
41
42               utils.show_mode_selection(Test,image,frame_width)
43
44              final_image = cv2.resize(image,(912,480),
    interpolation=cv2.INTER_AREA)
45              video_writer.write(final_image)
46
47              if Stream==1:
48                  cv2.imshow("Model Detection",final_image)
49                  utils.send_frame_to_server(final_image,SERVER_URL)
50
51              elif Stream==0:
52                  cv2.imshow("Model Detection",final_image)
53
54              key = cv2.waitKey(0) & 0xFF
55              if key == ord('c'):
56                  Test = 0
57                  hr_analysis_event.clear()
58              elif key == ord('h'):
59                  Test = 1
60                  hr_analysis_event.set()
61              elif key == ord('b'):
62                  Test = 2
63                  hr_analysis_event.set()
64              elif key == ord('r'):
65                  hr_analysis_event.clear()
66                  print("Reset in corso")
67                  utils.write_new_misuration(output_file_1,
    output_file_2,output_file_3,output_file_4)
68                  utils.clear_queue(bpm_queue)
69                  utils.clear_queue(data_queue)
70
71                  red_values_total.clear()
72                  green_values_total.clear()
73                  blue_values_total.clear()
74
75                  red_values_forhead.clear()
76                  green_values_forhead.clear()
77                  blue_values_forhead.clear()
78
79                  red_values_left.clear()
```

```
                green_values_left.clear()
                blue_values_left.clear()

                red_values_right.clear()
                green_values_right.clear()
                blue_values_right.clear()

                if Test==1 or Test==2:
                    print("Reset completato")
                    hr_analysis_event.set()
                else:
                    print("Reset completato")
            elif key == ord('s'):
                if Stream==0:
                    Stream=1
                else:
                    Stream=0

            elif key == ord('q'):  # Premere 'q' per uscire
                break
```

The capability of the program to change mode of execution is possible because the `Test` variable is dynamic.In fact with `waitKey()` the program wait for a certain key and pressing a key from `C`, `H`, and `B` switches between the modes in real time, and pressing `Q` stops the execution. As you can see above this switch between the keys `C`, `H`, `B` activates and deactivates also the HR event. Therefore, when the `Test` variable changes from 0 to 1 or 2, the `analyze_heartbeat` thread will not wait anymore, so the HR BLOCK 2 will be executed.

```
def analyze_heartbeat(data_queue, bpm_queue, time_queue):
    while True:
        hr_analysis_event.wait()
        # HR BLOCK 2
```

Thread execution mode is managed by the following code, where the `analyze_heartbeat` thread is set as a daemon to ensure that pressing the `Q` key stops both threads and terminates the script.

There are also two other modes, which we could define as "ghost mode": the streaming mode and the reset mode. Pressing the **S** key

will change the default value of the *Streaming* variable from 0 to 1 (and vice versa). In this way, every frame processed by the code will be both displayed using the `imshow()` function and sent to a Flask server, which will stream the video on a local network. We will look at this implementation in detail in a later paragraph.

Pressing the **R** key, instead, activates the reset mode, which, as you can see from the previous code, deletes all queues and variables containing raw values useful for heart rate estimation, allowing for a new measurement.

Essentially, this portion of code:

- disables the `hr_analysis()` event to prevent multiple portions of code from accessing resources simultaneously
- deletes queues and variables containing values for heart rate estimation
- checks the `Test` variable and, consequently, re-enables or keeps disabled the `hr_analysis()` event

```
1 if __name__ == '__main__':
2     Thread(target=main, args=(cap, data_queue, bpm_queue,
    time_queue), daemon=False).start()
3     Thread(target=analyze_heartbeat, args=(data_queue, bpm_queue,
    time_queue), daemon=True).start()
```

2.3 HR/Age/Gender Recognition

In this section we analyze the code used for heart rate estimation. The code is structured into two main parts, **HR BLOCK 1** and **HR BLOCK 2**, corresponding to the `HR` and `Both` modes (with the test variable set to 1 and 2, respectively). We follow the exact execution order, starting with the main thread (`main`) and then examining the

secondary thread (`analyze_heartbeat`), which runs in parallel with the main thread as explained in the previus chapter.

First, facemesh is computed using the MediaPipe framework, which is later used to extract the necessary facial regions for heart rate estimation. MediaPipe Face Mesh estimates 468 3D facial landmarks in real time by employing machine learning to infer the 3D facial surface, requiring only a single camera input (without the need for a dedicated depth sensor)[16]. This makes it an excellent solution for our application.

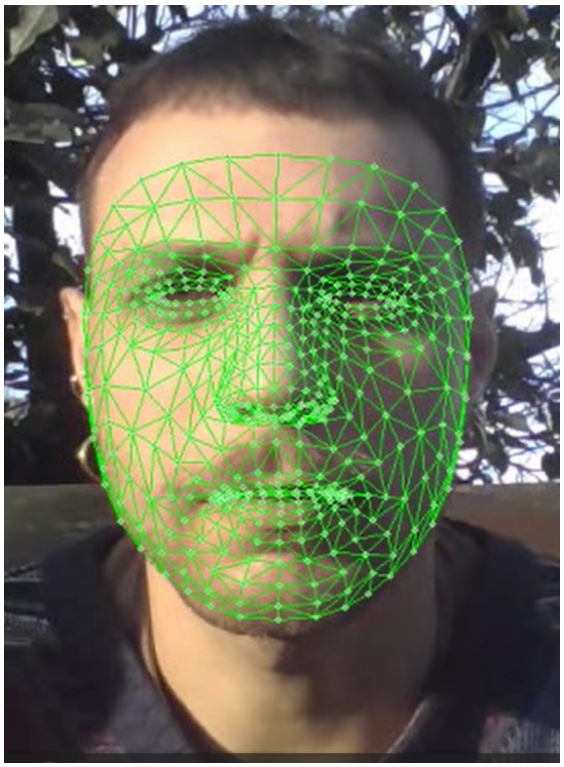


Figure 2.2: Face mesh applied on the real-time webcam feed

At this point, the dimensions of the frame are extracted to determine the actual positions of the landmarks. Since MediaPipe returns normalized values between 0 and 1, these are multiplied by the frame dimensions to obtain the absolute positions[17].

```
if results.multi_face_landmarks:
    # checks if the list is empty or not
    for face_landmarks in results.multi_face_landmarks:
    # iterates over each face found in the landmarks list

        # fd.show_facemesh(image, mp_drawing, face_landmarks,
    mp_face_mesh)

        img_height, img_width, _ = image.shape
        x_coords = []
        y_coords = []

        for landmark in face_landmarks.landmark:  # iterates over
    all landmarks of a single face
            x = int(landmark.x * img_width)         # multiplies
    by the image dimensions since values are normalized
            y = int(landmark.y * img_height)        # multiplying
    by the image dimensions yields the absolute position
            x_coords.append(x)
            y_coords.append(y)
```

Using the minimum and maximum landmark values (with a tolerance), a cropped image is extracted for further processing:

```
cropped_image = image[y_min:y_max, x_min:x_max]
```

The cropped image is then passed to the `extract_heartbeat` function which extracts the mean values of each color channel (red, green, and blue) for the forehead, left cheek, and right cheek using three differents masks [18] And a fourth mask that sums the three as a comparison, the total mask. The choice to use these three Regions Of Interest (ROI) is due to the fact that these areas are less prone to movement due to the facial expression, have a high blood perfusion (also influenced by the reduced size of the tissues)[19], lower interference from shadows and hair, and a uniform distribution of light.

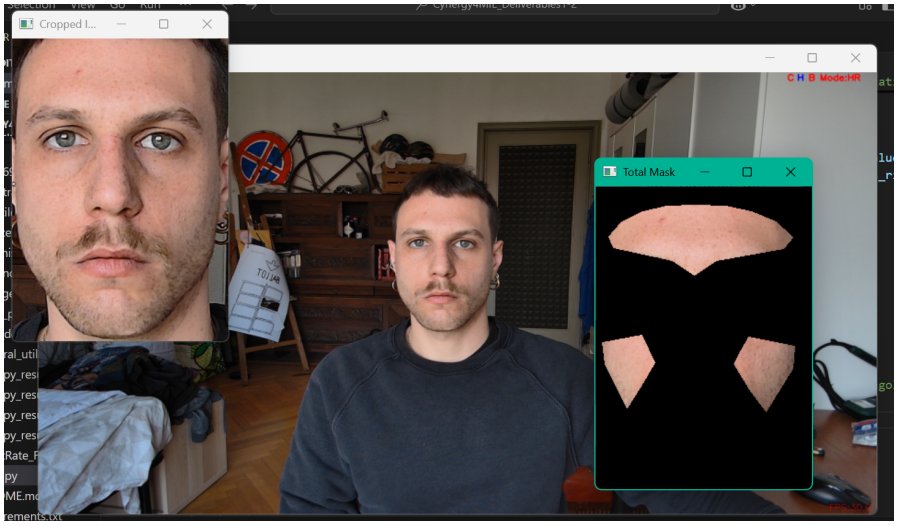


Figure 2.3: The extraction of the cropped image and facial portions in real-time

```
def extract_heartbeat(image, face_landmarks, width, height, x_min, y_min):
    # Indices for specific regions
    forehead_indices = [54, 103, 67, 109, 10, 338, 297, 332, 284, 298, 336, 8, 107, 68]  # see image
    right_cheek_indices = [330, 266, 423, 436, 434, 376, 352]
    left_cheek_indices = [101, 36, 203, 216, 214, 147, 123]

    # Obtain the contours of the forehead
    forehead_points = []
    right_cheek_points = []
    left_cheek_points = []

    for i in forehead_indices:
        x = int(face_landmarks.landmark[i].x * width) - x_min
        y = int(face_landmarks.landmark[i].y * height) - y_min
        forehead_points.append((x, y))

    for i in right_cheek_indices:
        x = int(face_landmarks.landmark[i].x * width) - x_min
        y = int(face_landmarks.landmark[i].y * height) - y_min
        right_cheek_points.append((x, y))

    for i in left_cheek_indices:
        x = int(face_landmarks.landmark[i].x * width) - x_min
        y = int(face_landmarks.landmark[i].y * height) - y_min
        left_cheek_points.append((x, y))

```

```
27      # Create a mask for the forehead
28      forehead_mask = np.zeros(image.shape[:2], np.uint8)
29      cv2.fillPoly(forehead_mask, [np.array(forehead_points, np.
        int32)], 255)
30
31      right_cheek_mask = np.zeros(image.shape[:2], np.uint8)
32      cv2.fillPoly(right_cheek_mask, [np.array(right_cheek_points,
        np.int32)], 255)
33
34      left_cheek_mask = np.zeros(image.shape[:2], np.uint8)
35      cv2.fillPoly(left_cheek_mask, [np.array(left_cheek_points, np.
        int32)], 255)
36
37      combined_mask = cv2.bitwise_or(forehead_mask, right_cheek_mask
        )
38      combined_mask = cv2.bitwise_or(combined_mask, left_cheek_mask)
39
40      # Apply the masks to the image
41      masked_forehead = cv2.bitwise_and(image, image, mask=
        forehead_mask)
42      masked_right_cheek = cv2.bitwise_and(image, image, mask=
        right_cheek_mask)
43      masked_left_cheek = cv2.bitwise_and(image, image, mask=
        left_cheek_mask)
44      total_mask = cv2.bitwise_and(image, image, mask=combined_mask)
45
46      # Calculate the mean RGB values for each region
47      mean_forehead = cv2.mean(masked_forehead, mask=forehead_mask)
        [:3]
48      mean_right_cheek = cv2.mean(masked_right_cheek, mask=
        right_cheek_mask)[:3]
49      mean_left_cheek = cv2.mean(masked_left_cheek, mask=
        left_cheek_mask)[:3]
50      mean_combined = cv2.mean(total_mask, mask=combined_mask)[:3]
51
52      cv2.imshow("Total Mask", total_mask)
53
54      return mean_combined, mean_forehead, mean_right_cheek,
        mean_left_cheek
```

Next, a timestamp is recorded for the measurement so that the algorithm's output can later be compared with that of a wrist heart rate monitor. The mean values are used to populate data lists using `upload_hr_lists` and to limit their length to 300 values via `limit_hr_list`:

```
def upload_hr_lists(mean_total, mean_forehead, mean_right, mean_left,
                    red_values_total, green_values_total, blue_values_total,
                    red_values_forhead, green_values_forhead, blue_values_forhead,
                    red_values_left, green_values_left, blue_values_left,
                    red_values_right, green_values_right, blue_values_right):

    r_total, g_total, b_total = mean_total
    r_forehead, g_forehead, b_forehead = mean_forehead
    r_right, g_right, b_right = mean_right
    r_left, g_left, b_left = mean_left

    # Total
    red_values_total.append(r_total)
    green_values_total.append(g_total)
    blue_values_total.append(b_total)
    # Forehead
    red_values_forhead.append(r_forehead)
    green_values_forhead.append(g_forehead)
    blue_values_forhead.append(b_forehead)
    # Left cheek
    red_values_left.append(r_left)
    green_values_left.append(g_left)
    blue_values_left.append(b_left)
    # Right cheek
    red_values_right.append(r_right)
    green_values_right.append(g_right)
    blue_values_right.append(b_right)
```

```
def limit_hr_list(red_values_total, green_values_total, blue_values_total,red_values_forhead, green_values_forhead, blue_values_forhead,red_values_left, green_values_left, blue_values_left,red_values_right, green_values_right, blue_values_right):

    if len(red_values_total) > 300:  # 10 seconds at 30 FPS
        red_values_total.pop(0)
        green_values_total.pop(0)
        blue_values_total.pop(0)

    if len(red_values_forhead) > 300:
        red_values_forhead.pop(0)
        green_values_forhead.pop(0)
```

```
11          blue_values_forhead.pop(0)
12
13      if len(red_values_left) > 300:
14          red_values_left.pop(0)
15          green_values_left.pop(0)
16          blue_values_left.pop(0)
17
18      if len(red_values_right) > 300:
19          red_values_right.pop(0)
20          green_values_right.pop(0)
21          blue_values_right.pop(0)
```

The limitation of the measurement values is performed by deleting the oldest value from the list. The choice of the 300 is to keep the necessary number of samples in memory without running the risk of an early deletion of the value before it is analyzed and making sure not to overload the machine to the detriment of the execution speed. Then data is then added to the queue for processing in the secondary thread:

```
1 data_queue.put((red_values_total, green_values_total,
      blue_values_total,red_values_forhead, green_values_forhead,
      blue_values_forhead,red_values_left, green_values_left,
      blue_values_left,red_values_right, green_values_right,
      blue_values_right))
```

Execution then shifts to `analyze_heartbeat` and, being in HR or Both execution mode the `hr_analysis_event` event will already be active and consequently the execution of `HR BLOCK 2` will occur without any problem. So, after verifying that there are at least 180 values for each color channel, the data is processed with `HeartPy` to compute the heart rate:

```
1 working_data_red_total, measures_red_total = hp.process(np.asarray
      (hr.bandpass(red_values_total)), sample_rate=30)
2 working_data_green_total, measures_green_total = hp.process(np.
      asarray(hr.bandpass(green_values_total)), sample_rate=30)
3 working_data_blue_total, measures_blue_total = hp.process(np.
      asarray(hr.bandpass(blue_values_total)), sample_rate=30)
4
5 working_data_red_forehead, measures_red_forehead = hp.process(np.
      asarray(hr.bandpass(red_values_forhead)), sample_rate=30)
```

```
working_data_green_forehead, measures_green_forehead = hp.process(
    np.asarray(hr.bandpass(green_values_forhead)), sample_rate=30)
working_data_blue_forehead, measures_blue_forehead = hp.process(np
    .asarray(hr.bandpass(blue_values_forhead)), sample_rate=30)

working_data_red_left, measures_red_left = hp.process(np.asarray(
    hr.bandpass(red_values_left)), sample_rate=30)
working_data_green_left, measures_green_left = hp.process(np.
    asarray(hr.bandpass(green_values_left)), sample_rate=30)
working_data_blue_left, measures_blue_left = hp.process(np.asarray
    (hr.bandpass(blue_values_left)), sample_rate=30)

working_data_red_right, measures_red_right = hp.process(np.asarray
    (hr.bandpass(red_values_right)), sample_rate=30)
working_data_green_right, measures_green_right = hp.process(np.
    asarray(hr.bandpass(green_values_right)), sample_rate=30)
working_data_blue_right, measures_blue_right = hp.process(np.
    asarray(hr.bandpass(blue_values_right)), sample_rate=30)
```

This processing is done through `HeartPy`, which through the `hp.process` function, gives us the average heart rate of the considered frame. Its sample has been set to 30 Hz corresponding to the FPS of the video signal. The average values of the color each channel is then filtered through a band pass filter with the cutoff frequencies set between 0.7 and 2 Hz which would allow an HR value measurement ranging from 42 to 120 bpm, subsequently these values are passed to a low pass filter with a cutoff frequency of 0.9 Hz in order to remove sudden signal variations at very high frequencies.

```
def bandpass(data, fs=20, order=2, fc_low=0.7, fc_hig=2,
    fc_lowpass=0.9):
     nyq = 0.5 * fs  % Nyquist frequency
     cut_low = fc_low / nyq  % Low cutoff frequency
     cut_hig = fc_hig / nyq  % High cutoff frequency

     # Design of the bandpass filter
     bp_b, bp_a = sig.butter(order, (cut_low, cut_hig), btype='
    bandpass')

     # Low-pass filter to remove very high frequencies (optional)
     lowpass_b, lowpass_a = sig.butter(order, fc_lowpass / nyq,
    btype='low')

```

```
12      # Apply the bandpass filter
13      filtered_data = sig.filtfilt(bp_b, bp_a, data)
14
15      # Apply the low-pass filter (if necessary)
16      filtered_data = sig.filtfilt(lowpass_b,lowpass_a,
        filtered_data)
17
18      return filtered_data
```

Since these filters are digital, to avoid aliasing effect a lowpass filter is needed. The filters were implemented using second order `Butterworth` filters for their minimal signal distortion and smooth, ring-free response. These were applied to the data via `filtfilt`, then filtering back and forth to avoid signal delays and thus maintain temporal alignment with the timestamp which is critical for testing the results. The newly obtained heart rate values were then loaded into the previously created csv files corresponding to the time values and added to the `bpm_queue` queue in order to allow the main thread to draw the results on the image frame by frame.

```
1  with open(output_file_1, mode='a', newline='') as file:
2      writer = csv.writer(file)
3      writer.writerow([date_list, time_list, measures_green_total['
       bpm'], measures_blue_total['bpm'], measures_red_total['bpm']])
4
5  with open(output_file_2, mode='a', newline='') as file:
6      writer = csv.writer(file)
7      writer.writerow([date_list, time_list, measures_green_forehead
       ['bpm'], measures_blue_forehead['bpm'], measures_red_forehead['
       bpm']])
8
9  with open(output_file_3, mode='a', newline='') as file:
10     writer = csv.writer(file)
11     writer.writerow([date_list, time_list, measures_green_left['
       bpm'], measures_blue_left['bpm'], measures_red_left['bpm']])
12
13 with open(output_file_4, mode='a', newline='') as file:
14     writer = csv.writer(file)
15     writer.writerow([date_list, time_list, measures_green_right['
       bpm'], measures_blue_right['bpm'], measures_red_right['bpm']])
16
```

```
bpm_queue.put((measures_green_total['bpm'], measures_blue_total['bpm'], measures_red_total['bpm'],
               measures_green_forehead['bpm'], measures_blue_forehead['bpm'], measures_red_forehead['bpm'],
               measures_green_left['bpm'], measures_blue_left['bpm'], measures_red_left['bpm'],
               measures_green_right['bpm'], measures_blue_right['bpm'], measures_red_right['bpm']))  % added as tuple
```

Finally, the heartbeat data is drawn on the image under the "box" drawn by the function `draw_fancy_box`

```
def draw_BPM(bpm_queue, image, x_min, y_max):
    try:
        bpm = bpm_queue.get_nowait()  % Non-blocking, raises an exception if the queue is empty
        bpm_g_t = bpm[0]
        bpm_b_t = bpm[1]
        bpm_r_t = bpm[2]
        bpm_g_f = bpm[3]
        bpm_b_f = bpm[4]
        bpm_r_f = bpm[5]
        bpm_g_l = bpm[6]
        bpm_b_l = bpm[7]
        bpm_r_l = bpm[8]
        bpm_g_r = bpm[9]
        bpm_b_r = bpm[10]
        bpm_r_r = bpm[11]

        cv2.putText(image, f"T BPM: g:{int(bpm_g_t)} b:{int(bpm_b_t)} r:{int(bpm_r_t)}", (x_min+5, y_max-10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 0, 0), 2)
        cv2.putText(image, f"F BPM: g:{int(bpm_g_f)} b:{int(bpm_b_f)} r:{int(bpm_r_f)}", (x_min+5, y_max+10), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)
        cv2.putText(image, f"L BPM: g:{int(bpm_g_l)} b:{int(bpm_b_l)} r:{int(bpm_r_l)}", (x_min+5, y_max+40), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 0, 255), 2)
        cv2.putText(image, f"R BPM: g:{int(bpm_g_r)} b:{int(bpm_b_r)} r:{int(bpm_r_r)}", (x_min+5, y_max+60), cv2.FONT_HERSHEY_SIMPLEX, 0.7, (255, 255, 255), 2)
```

```
def draw_fancy_box(frame, x_min, x_max, y_min, y_max, radius=20, color=(255, 255, 255), thickness=2):
    cv2.ellipse(frame, (x_min + radius, y_min + radius), (radius, radius), 180, 0, 90, color, thickness)
    cv2.ellipse(frame, (x_max - radius, y_min + radius), (radius, radius), 270, 0, 90, color, thickness)
    cv2.ellipse(frame, (x_min + radius, y_max - radius), (radius, radius), 90, 0, 90, color, thickness)
    cv2.ellipse(frame, (x_max - radius, y_max - radius), (radius, radius), 0, 0, 90, color, thickness)
```

Age and gender are estimated using the `age_gender_recognition` function, which leverages a pre-defined model:

```
def age_gender_recognition(frame, image, age, gender, y_min, y_max, x_min, x_max):
    try:
        face_crop = frame[y_min:y_max, x_min:x_max]
        face_blob = cv2.dnn.blobFromImage(face_crop, 1.0, (227, 227), MEAN_VALUES, swapRB=False)

        gender.setInput(face_blob)
        gender_preds = gender.forward()
        gender_out = gender_classification[gender_preds[0].argmax()]

        # Age recognition
        age.setInput(face_blob)
        age_preds = age.forward()
        age_out = age_classifications[age_preds[0].argmax()]

        # Display age and gender on the face
        label = f"{gender_out}, {age_out}"
        cv2.putText(image, label, (x_min, y_min - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)
    except:
        print('Error')
```

Finally, the following classifications and mean values are used:

```
% CLASSIFICATION SETUP
age_classifications = ['(0-2)', '(4-6)', '(8-12)', '(15-20)', '
    (22-32)', '(38-43)', '(48-53)', '(60-100)']
gender_classification = ['M', 'F']

MEAN_VALUES = [114, 117, 123]
```

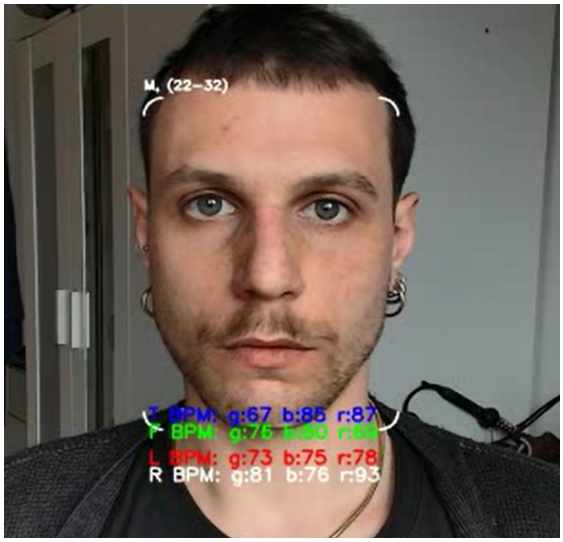


Figure 2.4: Heart Rate values, fancybox and age/gender recognition drawings

2.4 People counting and body pose detection

In this section, we will analyze the part of the code that deals with **people counting**, which we previously defined as **BODYPOSE/PEOPLE COUNTING BOX**. This is the portion of the code that is activated in **People Counting mode** (the default mode) or in **BOTH mode**.

This portion of the code has three modes too, which can be activated using three boolean variables:

```
SHOW_BBOX = False
SHOW_KEYPOINTS = False
```

```
3 SHOW_BOTH = True
```

These variables must be set so that only one of them is `True`, while the other two must be `False`. Specifically, the three modes perform the following functions:

- **SHOW_BBOX**: Displays only the bounding box and the confidence score with which YOLO recognizes a person.
- **SHOW_KEYPOINTS**: Displays only the keypoints for each detected body.
- **SHOW_BOTH**: Displays the bounding box, confidence score, and keypoints with their connections, forming what is called a *"skeleton"*.

The **confidence score** represents how certain the model is that the detected element in the image is a person. This value ranges from 0 to 1.

Once the mode is set, the actual frame prediction takes place:

```
1 results = model(image, conf=0.45, classes=[0], verbose=False)
```

This function performs the image prediction (in this case, a video frame) with specific parameters for our application:

- `image` is the frame to be analyzed.
- `conf=0.45` is the minimum confidence threshold we set. The algorithm will only detect and count people if their confidence score is above this value.
- `classes=[0]` ensures that the model predicts **only people**, since YOLO is designed to recognize a wide range of objects.
- `verbose=False` prevents the terminal from displaying a message for each frame prediction.

At this point, we have a series of `if` and `else` statements that allow the code to execute in the desired mode.

The code checks whether the variable `SHOW_BOTH` is set to `True`. If so, it draws both the bounding box and the skeleton on the frame:

```
results = model(image, conf=0.45, classes=[0], verbose=False)
```

If `SHOW_BOTH` is `False`, execution moves to another section of the code, where the bounding box coordinates and confidence score are first extracted:

```
for result in results:
    for box in result.boxes:
        x1, y1, x2, y2 = map(int, box.xyxy[0])
        conf = box.conf[0].item()
```

Depending on the selected mode, the bounding box and confidence score will be drawn if `SHOW_BBOX=True`, and the keypoints will be drawn if `SHOW_KEYPOINTS=True`.

```
# Draw BBOX
if SHOW_BBOX:
    cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
    label = f"Person {conf:.2f}"
    cv2.putText(annotated_frame, label, (x1, y1 - 10),
                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

# Draw KEYPOINTS
if SHOW_KEYPOINTS:
    if SHOW_KEYPOINTS:
        keypoints_array = result.keypoints.xy
            for person_keypoints in keypoints_array:
                filtered_keypoints = [(int(x), int(y)) for x, y in person_keypoints if (x, y) != (0, 0)]
                    for x, y in filtered_keypoints:
                        cv2.circle(annotated_frame, (x, y), 5, (0, 0, 255), -1)
```

As seen in the code above, keypoints at position `(0,0)` are removed.

This is due to a **YOLO bug**, where manually extracted keypoints always include one fixed at `(0,0)`.

Regarding **people counting**, this process occurs regardless of the execution mode. The algorithm counts the **bounding boxes** detected by the model and displays the count at the **top-left corner** of the frame.

Figure 2.5: Output of SHOW_BBOX mode

Figure 2.6: Output of SHOW_KEYPOINTS mode

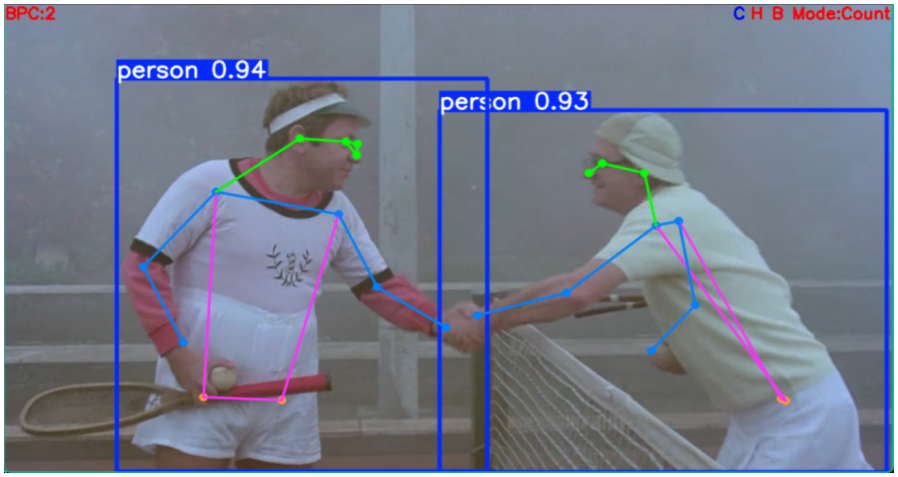


Figure 2.7: Output of SHOW_BOTH mode

2.5 Video Streaming

As previously mentioned, the script includes a **"ghost mode"**, called **"Streaming mode"**, which allows the program to display video frames using the `imshow()` function while simultaneously transmitting them in **streaming** over a local network.

This functionality has been implemented using a **Flask server**, a lightweight and modular web application written in Python, which handles HTTP requests and enables web application development.

By pressing the **S** key, the program activates the `send_frame_to_server()` function, which sends the video frames to the previously defined server URL.

The server is configured to receive, process, and transmit frames as follows:

```
1 from flask import Flask, request, send_file, Response, abort
2 from flask_httpauth import HTTPBasicAuth
3 import cv2
4 import numpy as np
5 import io
6
7 app = Flask(__name__)
```

```
auth = HTTPBasicAuth()


USERNAME = 'user'
PASSWORD = 'tigre'

latest_frame = None

# per verificare le credenziali
@auth.verify_password
def verify_password(username, password):
    if username == USERNAME and password == PASSWORD:
        return True
    return False

@app.route('/upload_frame', methods=['POST'])
def upload_frame():
    global latest_frame
    file = request.files['file']
    file_bytes = np.frombuffer(file.read(), np.uint8)
    latest_frame = cv2.imdecode(file_bytes, cv2.IMREAD_COLOR)
    return "Frame ricevuto", 200

@app.route('/latest_frame')
@auth.login_required  # Protezione con autenticazione
def get_latest_frame():
    global latest_frame
    if latest_frame is None:
        return "Nessun frame ricevuto", 404
    _, buffer = cv2.imencode('.jpg', latest_frame)
    return send_file(
        io.BytesIO(buffer),
        mimetype='image/jpeg',
        as_attachment=False
    )


def generate_frames():
    global latest_frame
    while True:
        if latest_frame is not None:
            ret, buffer = cv2.imencode('.jpg', latest_frame)
            if ret:
                frame = buffer.tobytes()
                yield (b'--frame\r\n'
                       b'Content-Type: image/jpeg\r\n\r\n' + frame
    + b'\r\n\r\n')


```

```
56 @app.route('/video_feed')
57 @auth.login_required  # Protezione con autenticazione
58 def video_feed():
59     return Response(generate_frames(), mimetype='multipart/x-mixed
   -replace; boundary=frame')
60
61 if __name__ == '__main__':
62     app.run(host='0.0.0.0', port=5000, debug=True)
```

As seen above, the frame is encoded and sent to the server. Anyone connected to the same network and aware of the server URL can access the stream and view the video in real time.

Additionally, to ensure a basic level of security, a simple authentication system with **username and password** has been implemented, allowing only authorized users to access the streaming service.

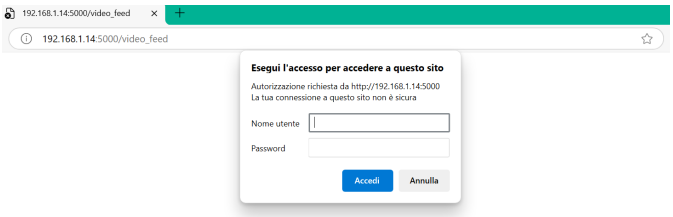


Figure 2.8: Autentication system

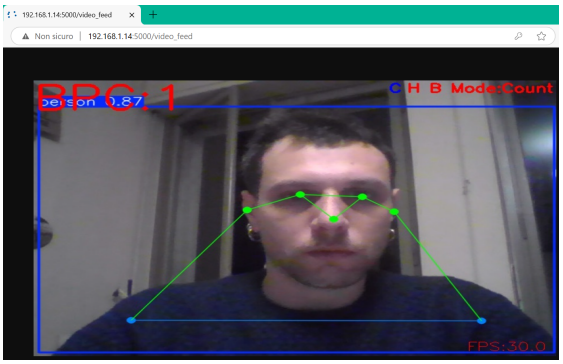


Figure 2.9: Video streaming

2.6 Raspberry Pi Adaptation

The code just explained was initially written and tested with a USB webcam on a Windows Machine, but to make it work on the raspberry Pi with a Linux Os, some modifications were necessary. USB webcams on Linux work through a generic USB driver like v4l2 [20], while Raspberry Pi cameras work through one of the two CSI (Camera Serial Interface) serial ports [21]. As a result, the code was modified using `Picamera2`, an integrated module with Raspberry Pi, to work with official cameras to obtain the frame, while the rest of the processing remained unchanged using OpenCV for all the image elaboration. We then installed and imported Picamera2 and replaced this portion of code from this:

```
cap=cv2.VideoCapture(0)
cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 1)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 2048)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 1080)
```

to this:

```
picam2 = Picamera2()
picam2.preview_configuration.main.size = (912, 480)
picam2.preview_configuration.main.format = "RGB888"
picam2.start()
```

Thus, activating the camera before the actual execution begins.
In the main thread, we had to change the extraction of the frame from this:

```
ret, frame = cap.read()
```

to this:

```
frame = picam2.capture_array()
```

Of course we had to we had to delete all the references of the code to `"cap"` to make the code work flawless.

Chapter 3

Testing

3.1 Introduction

In this chapter, we will describe the tests performed on the device as a whole, starting from the software and proceeding to the verification of the device's operation when powered solely by a power bank.

As already described in the previous chapter, the script operates in three distinct modes: **People Counting**,**HR and Age/Gender Detection** , **Both** . However, since the Both mode is essentially the simultaneous execution of the first two, in the first section, we will analyze only the two main modes.

In the second section, we will examine how the device runs the script while being powered solely by a power bank and what strategies have been adopted to ensure its proper functioning, thus making the device self-sufficient.

3.2 Software

Here, we will analyze how the tests of the two previously described modes were conducted, detailing the testing procedures, the conditions under which they were performed, and the results obtained. These tests were carried out both on the test machine and on the device,

consistently yielding the same results due to the shared code.

3.2.1 Age/Gender Detection

For the **HR and Age/Gender Detection** testing, it was necessary to use a specific dataset. In this case, the choice fell on UTKFace, a dataset consisting of 23,761 JPEG images containing human faces with ages ranging from 0 to 116 years [22]. To perform the test, we wrote a dedicated script that follows the same execution pattern as the main code, using MediaPipe for face extraction and `.Prototxt` and `.CaffeModel` for performing the estimation in the age and gender classes. The only difference compared to the main script concerns the input: in this code, OpenCV does not access the camera feed to perform the predictions, but the code is placed inside a for loop to process all the images in the dataset.

```
1  import cv2
2  import os
3  import glob
4  import pandas as pd
5  import mediapipe as mp
6
7  age_classifications = ['(0-2)', '(4-6)', '(8-12)', '(15-20)', '
       (22-32)', '(38-43)', '(48-53)', '(60-100)']
8  gender_classification = ['M', 'F']
9  MEAN_VALUES = [114, 117, 123]
10
11 age_prototxt = "Models/Gender_Age_Models/deploy_age2.prototxt"
12 age_model = "Models/Gender_Age_Models/age_net2.caffemodel"
13 gender_prototxt = "Models/Gender_Age_Models/gender_deploy.prototxt
       "
14 gender_model = "Models/Gender_Age_Models/gender_net.caffemodel"
15
16 age_net = cv2.dnn.readNet(age_model, age_prototxt)
17 gender_net = cv2.dnn.readNet(gender_model, gender_prototxt)
18
19 mp_face_detection = mp.solutions.face_detection
20 face_detection = mp_face_detection.FaceDetection(
       min_detection_confidence=0.5)
21
22
```

```
23 def process_utkface_dataset(dataset_path, output_file):
24     image_paths = glob.glob(os.path.join(dataset_path, "*.*"))
25     results = []
26     i = 0
27 
28     for image_path in image_paths:
29         image = cv2.imread(image_path)
30         if image is None:
31             continue
32 
33         filename = os.path.basename(image_path)
34         h, w, _ = image.shape
35 
36         image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
37         results_faces = face_detection.process(image_rgb)
38 
39         if results_faces.detections:
40             for detection in results_faces.detections:
41                 bboxC = detection.location_data.
    relative_bounding_box
42                 x, y, width, height = int(bboxC.xmin * w), int(
    bboxC.ymin * h), int(bboxC.width * w), int(bboxC.height * h)
43                 x, y = max(0, x), max(0, y)
44                 face_crop = image[y:y+height, x:x+width]
45 
46                 try:
47                     face_blob = cv2.dnn.blobFromImage(face_crop,
    1.0, (227, 227), MEAN_VALUES, swapRB=False)
48 
49                     gender_net.setInput(face_blob)
50                     gender_preds = gender_net.forward()
51                     gender_out = gender_classification[
    gender_preds[0].argmax()]
52 
53                     age_net.setInput(face_blob)
54                     age_preds = age_net.forward()
55                     age_out = age_classifications[age_preds[0].
    argmax()]
56 
57                     results.append([filename, gender_out, age_out
    ])
58                     i += 1
59                     print(f"Elaborata immagine {i}")
60                 except:
61                     print("Errore elaborazione immagine")
62                     results.append([filename, "Error", "Error"])
63                     continue
64         else:
65             print(f"Nessun volto rilevato in: {filename}")
```

```
66                results.append([filename, "No face", "No face"])
67
68        df = pd.DataFrame(results, columns=["Filename", "Gender", "Age
          "])
69        df.to_excel(output_file, index=False)
70
71    if __name__ == "__main__":
72        dataset_path = "UTKFace"
73        output_file = "utkface_results.xlsx"
74        process_utkface_dataset(dataset_path, output_file)
```

Moreover, as seen in the code, the results are inserted into an Excel file structured with `[filename],[gender_prediction],[age_prediction]` to allow comparison of the results. The decision to include the filename in the output file is due to the fact that each image in the dataset is named in the format `[age][gender][race]_[date&time].jpg`[22]. In this way, with a small reformatting of the data, it became possible to compare the dataset information with the results obtained from the model and, accordingly, extract the results and the following confusion matrices.

Presumed Age Class

Real Age Class	(0-3)	(4-7)	(8-14)	(15-21)	(22-37)	(38-47)	(48-59)	(60-116)
(0-3)	1327	304	209	2	14	7	1	0
(4-7)	212	239	230	3	37	9	5	0
(8-14)	75	245	527	17	115	27	9	1
(15-21)	60	132	571	85	598	102	20	2
(22-37)	306	673	2925	309	4987	1185	173	18
(38-47)	63	99	528	39	1144	516	112	10
(48-59)	84	111	471	29	871	672	312	44
(60-116)	204	94	607	14	481	572	469	232

Figure 3.1: Age recognition confusion matrix

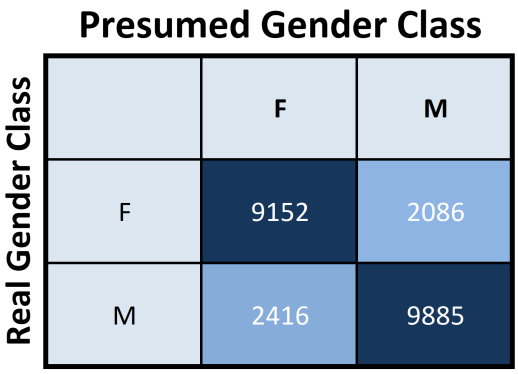


Figure 3.2: Gender recognition confusion matrix

From the **confusion matrix** related to **Age/Gender measurements**, we can observe that most of the high values are located along the diagonal, indicating that the model is fairly accurate, although the presence of errors is not negligible. It can also be noted that many errors are mainly distributed in adjacent classes. For example, individuals in the **(4-7)** class are often confused with **(0-3)** and **(8-14)**, while individuals in the **(22-37)** class are incorrectly classified as **(38-47)**.

This type of error may be due to the fact that aging does not follow strict boundaries. Consequently, it is difficult for a machine learning model to distinctly characterize the facial features of adjacent age classes.

We also notice that the **(60-116)** class is often confused with **(48-59)** or **(38-47)**, which could mean that the model does not recognize distinctive features of very elderly individuals.

From the **gender detection confusion matrix**, we can conclude that the prediction is significantly more stable. The model has an accuracy of around 80%, and we observe a clear difference between the

diagonal values and the other entries. This result was expected, as facial differences between male and female subjects are more pronounced, and there is a drastic reduction in the number of classes compared to age classification.

3.2.2 Body Pose /People Counting

For the people counting test, we directly used the main program, modifying the code segment responsible for video frame acquisition and providing a 720p video file as input.

To evaluate the accuracy and reliability of body pose estimation and people counting algorithms, we used two tennis match videos with different characteristics:

- A match from the **2012 Australian Open**, characterized by professional recordings and fairly stable shots, ideal for evaluating the model's behavior under optimal conditions.
- A scene from a film by **Paolo Villaggio** [23], with less structured recordings, lower resolution and overall video quality, and unconventional angles, useful for testing the algorithm's robustness under non-standard and more complex conditions.

The tests were repeated using three variants of the same YOLOv11-Pose model:[24]:

- **YOLOv11-n (Nano)**: the lightest and fastest model, specifically designed for devices with limited computational resources, such as embedded systems or edge devices.
- **YOLOv11-s (Small)**: a compromise between speed and accuracy, more powerful than YOLO-N but with excellent inference speed, making it perfect for scenarios requiring a good balance between performance and resource usage.

- **YOLOv11-m (Medium)**: a much more accurate model compared to YOLOv11-s, with more parameters and higher detection capability, but at a significant computational cost.

To improve inference speed on devices with limited computing power, such as our Raspberry Pi, we exported the model in **NCNN** format. This is a framework developed by Tencent for high-performance neural network inference computation, specifically optimized for mobile platforms.[25]

The choice to use **NCNN** over other frameworks is due to its significantly higher inference speed compared to competing frameworks, as illustrated in the following graph.[26]

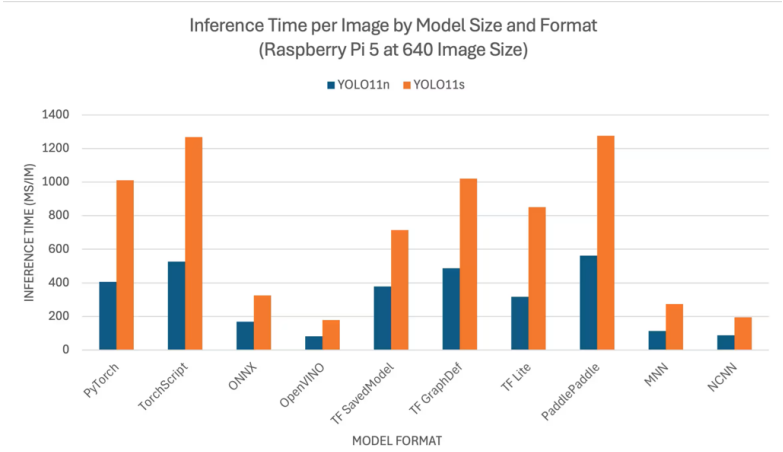
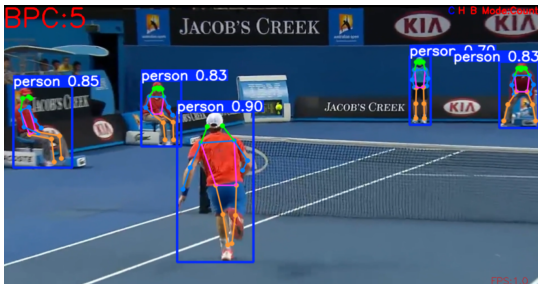


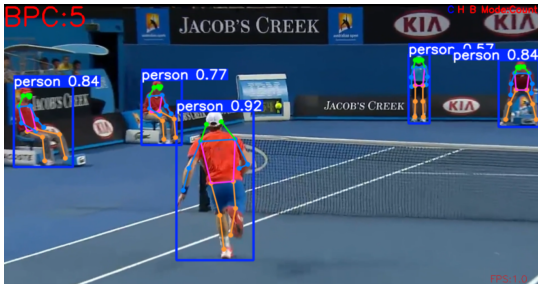
Figure 3.3: Graph of inference times of different YOLO-compatible frameworks

Regarding the test results: In the **Australian Open** video, all three models perform very well. The number of people and the body poses found are practically the same, although there is an obvious difference in measurement precision between the three models.

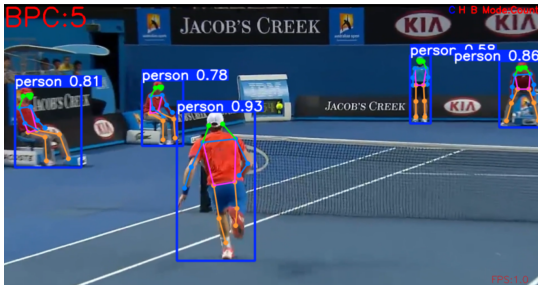
Also in the video excerpt from the **Paolo Villaggio** film, the three models perform very well. Although in the scene the players are always covered in fog, the models are always able to find the people present in the image



Result from YOLOv11-n

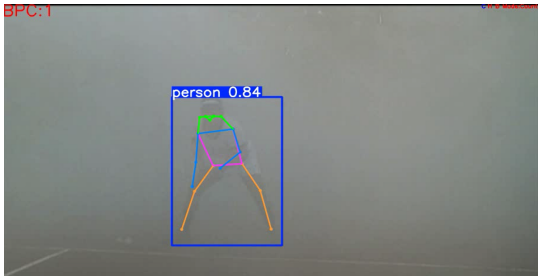


Result from YOLOv11-m

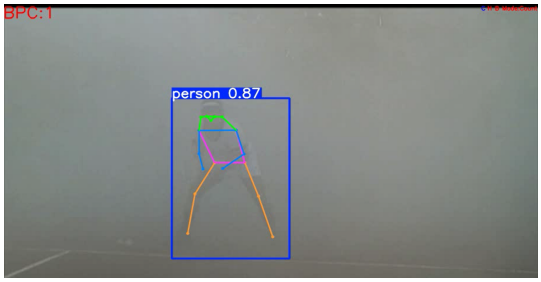


Result from YOLOv11-m

Result from YOLOv11-n



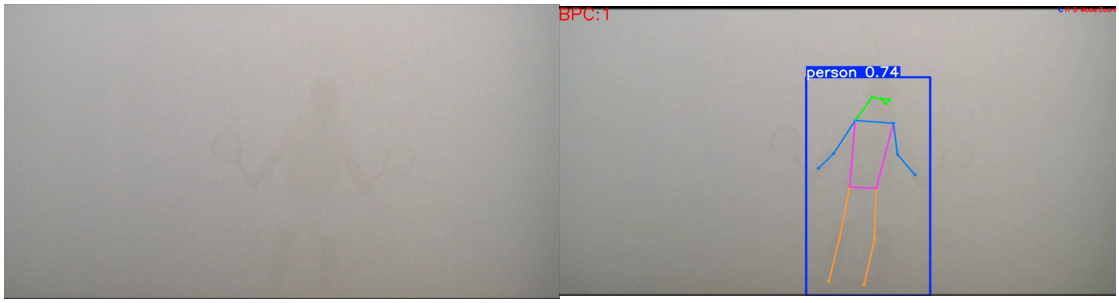
Result from YOLOv11-s



Result from YOLOv11-m

Figure 3.4: Comparison of the results of the Australian Open and the film

Furthermore, it is interesting to note that in a frame of this video, the fog is so thick that the person appears practically as a very light shadow, but the most powerful model that we're testing (YOLOv11-m) is still able to predict the position of the subject.



The player covered by the fog Result from YOLOv11-m

Figure 3.5: Comparison of the frame and the model applied

3.2.3 Heart Rate detection

For heart rate detection testing, we considered three main situations: **natural light**, **direct artificial light**, and **indirect artificial light**. As a result, we conducted heart rate detection tests with the three cameras chosen for the application, whose selection was explained in Chapter 1. Additionally, we performed a test with a fourth camera, a 4K USB camera with autofocus and auto-exposure, because this allowed us to reproduce the same three tests (natural light, direct artificial light, and indirect artificial light) both on the test machine and on the Raspberry Pi, in order to compare the results obtained.

The tests were all performed with the same subject stationary in front of the camera for a total of 5 minutes while wearing a **Garmin VIVO** on their wrist, which simultaneously measured the heart rate using **photoplethysmography (PPG)** through an optical sensor. The results were saved in a `.csv` file.

Regarding the test conditions:

- For the tests with **natural light**, we positioned a tripod near a window, trying as much as possible to achieve similar lighting conditions, particularly with a cloudy sky, to have the most diffused light possible. This was because direct light could have "burned" the image, thus distorting the measurement.

- For the tests **direct and indirect artificial light**, the camera was placed slightly in front of a warm light source: Facing the subject's face in the case of the direct artificial light test and pointed at a white wall for the indirect artificial light test, so as to diffuse the light evenly.

All the measurements performed were processed in post-processing to compare them with the values from the .csv file, the output of the Garmin. Here is an overview of what we did with the data obtained from the script:

- Created a pivot table to gather all the measurements from the same color channel related to the same second, so they could be averaged and compared to the measurements from the **Garmin**, which takes a reading every second.

- Applied a 5-second moving average to the averaged data to achieve more uniform measurements.

- Calculated a weighted average of the Red, Green, and Blue color channel results to obtain a smoother measurement, using **20%R-60%G-20%B** thus giving greater weight to the values extracted from the green channel [27].

  - The only exception was the **Raspberry HQ Camera**:
    Here we applied a different weighted average giving more importance to the Blue channel compared to the Red channel. This is because the camera has an IR filter, and consequently,

it tends to give more accurate values on the Blue channel. We therefore applied a weighted average of **20%R-20%G-60%B**.

- Finally, we compared the HR signal derived from the Garmin with our measurements to generate a comparison graph and reliability values computed respect our goal of 15 BPM maximum error.

Figure 3.6: Natural light test environment

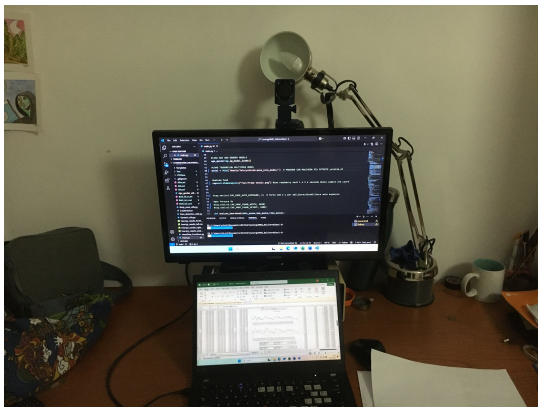


Figure 3.7: Direct artificial light environment

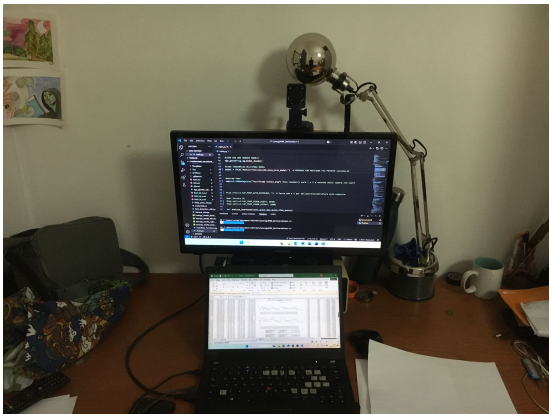


Figure 3.8: Indirect artificial light environment

Let's now look at some of the main numerical results of the tests we have performed. Each of the graphs drawn for each measurement are available in the appropriate appendices A,B,C and D.

USB Camera

In the table below, we can see the reliability values of the measurements performed by the script in the three light conditions tested using the same USB camera on both devices.

	Natural Light				Direct Artificial Light				Indirect Artificial Light			
	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total
PC	99,32%	96,95%	90,85%	87,80%	91,19%	83,05%	96,27%	96,95%	85,76%	88,14%	90,51%	84,07%
Raspberry	97,29%	96,95%	97,97%	96,61%	88,81%	89,49%	93,56%	97,97%	89,15%	97,63%	91,86%	82,03%

Figure 3.9: Reliability table of the misuration of the two devices

We can observe that, in general, both solutions perform well under these lighting conditions, with accuracy values consistently exceeding 80%. As expected, the highest accuracy values are obtained under natural light conditions, particularly in measurements taken on the forehead. Except for a few exceptions, both solutions seem to follow more or less the same pattern. Consequently, since their values are very similar, we can conclude that both solutions are reliable. We can also observe, from the graphs in Appendix A, that in both tests, the values obtained by the script follow those recorded by the **Garmin** with obvious exceptions due to the imperfect measurement environment

Raspberry Camera v3 and v3 wide

Just like in the previous paragraph, we analyze the numerical reliability results of these two cameras. We chose to discuss them together because they share the same sensor but not the same aperture (f) value.

	Natural Light				Direct Artificial Light				Indirect Artificial Light			
	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total
v3	77,97%	85,08%	89,49%	66,10%	78,31%	89,49%	87,12%	84,07%	50,51%	66,78%	80,34%	46,78%

Figure 3.10: Reliability table of the misuration of the camera v3

As shown in the tables, both cameras perform quite well under natural and direct artificial light conditions. However, the V3 has

	Natural Light				Direct Artificial Light				Indirect Artificial Light			
	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total
v3 Wide	89,49%	88,81%	92,54%	91,86%	76,61%	78,31%	82,03%	74,24%	94,92%	92,20%	89,49%	99,66%

Figure 3.11: Reliability table of the misuration of the camera v3

significantly lower accuracy values compared to the Wide under indirect artificial lighting.

Looking at the graphs in Appendices B and C, we can see that the values predicted by the script are always higher than the actual Garmin readings and those from the V3 Wide camera. This could be due to the fact that the V3 has a much lower aperture (f) value than the V3 Wide. Consequently, the error could be caused by a greater amount of light entering the sensor, despite the same lighting conditions used in the test.

Raspberry HQ Camera

As you can see from the table below, the camera in question appears to be the one with the highest accuracy values across all conditions

	Natural Light				Direct Artificial Light				Indirect Artificial Light			
	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total	Forehead	Left cheek	Right cheek	Total
HQ	96,61%	100,00%	100,00%	100,00%	87,46%	93,22%	86,44%	87,46%	93,22%	90,85%	95,59%	94,24%

Figure 3.12: Reliability table of the misuration of the HQ camera

In fact, along with the USB camera used and described earlier, it is one of the cameras with the highest sensor quality in terms of megapixels and lens.

Furthermore, by comparing the values obtained in the table with the graphs presented in Appendix D, this camera is the one whose values deviate the least from the Garmin measurement. Considering its weight and size, it seems to be the best candidate for our application.

3.3 Hardware Testing

Now that we have verified that all the code works correctly, we need to determine whether the device can actually function when powered by a battery, ensuring complete self-sufficiency. For this reason, we started by connecting the **power bank** to the **decoy** via a USB-C/USB-C cable, as explained in the first chapter, to request **12V** from the power bank. Then, we soldered two wires to the terminals of the **decoy** and connected the other ends to a **step-down converter**, allowing us to use its **5V USB-A output** to power the Raspberry via a **USB-A/USB-C cable**.

Figure 3.13: Test of the first chain

With this setup, the Raspberry was able to boot up and perform basic operations. However, as soon as it started the first inference using the **YOLO** model, the board would suddenly shut down. We immediately suspected that the issue was caused by the **overclocking** of the raspberry, which likely required more power than our power supply circuit could provide.
We performed this overclock to improve performance by simply modifying the config.txt configuration file. We then added 3 lines, namely: `force_turbo=1`, `arm_freq=2900`, and `GPU_freq=1000` and in the same way we can gradually **decrease the clock frequency** until we reached

the base frequency to check our theory, as shown in the table below:

CPU frequency (GHz)	GPU frequency (MHz)	Result
2900	800	Shuts down on first YOLO inference
2800	800	Shuts down on first YOLO inference
2700	800	Shuts down on first YOLO inference
2600	800	Shuts down on first YOLO inference
2500	800	Script starts but card goes into protection after a few seconds

Figure 3.14: Test with all the Clock frequency

As you can see, even at the default clock frequency, the board shut down shortly after the script started executing. We then connected an **oscilloscope** to the output of the **decoy** and observed that the voltage was stable, both on the **decoy** and on the **step-down converter**. However, as soon as the program started the first YOLO inference, the voltage dropped below **4.7V**. Even after attempting to modify the code to prevent the board from shutting down due to protection mechanisms, execution remained unstable. To address this, we tried connecting **two step-down converters in parallel** to ensure a **stable 5V supply** and then repeated the test.

Figure 3.15: Test with the stepdown connected in parallel

However, using the oscilloscope, we noticed that although the voltage was **5.1V**, the Raspberry was not requesting the necessary power from the **decoy PD 2.0**. As a result, the Raspberry continued shutting down during testing. Additionally, connecting two power supplies in parallel introduced a **ripple** of approximately **500mV**, which led us to seek another solution.

Figure 3.16: Ripple shown by the oscilloscope

Ultimately, we decided to **underclock** the Raspberry, reducing the CPU frequency to **2GHz**, compared to its default **2.5GHz** and **2.9GHz in overclocked mode**. With this configuration, we managed to keep the board running in **BOTH mode**, simultaneously executing **YOLO, FaceMesh**, and models for **age, gender, and heart rate estimation**, while also streaming the video output via a **Flask server** running directly on the Raspberry.All this only connecting the Raspberry **directly to the power bank**, removing all intermediary circuits, achieving a stable and fully functional system.

Figure 3.17: Raspberry connected to the only power bank

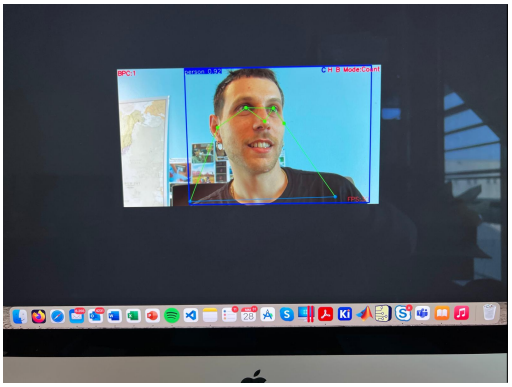


Figure 3.18: The output video of the Raspberry streamed

Conclusions

In this thesis project, we focused on the development of a device capable of detecting physiological values of individuals during search and rescue (SAR) missions. We first analyzed and selected the necessary components, considering weight, cost, and performance, and then developed a script capable of estimating age, biological gender, and heart rate. Additionally, the system can detect both stationary and moving people in real time, with the possibility of streaming the output to a local network.

The results we obtained were reasonably good, particularly in terms of body counting and body detection, biological gender estimation (with positive estimates above 80

Although the results obtained could have significant applications in SAR missions, helping estimate the number of people, their age, biological gender, and health status, they are not without limitations.

Indeed, at the current state of the art, the age detection model may not be accurate enough, and the algorithm's ability to estimate the heart rate of a single subject might not be sufficient for real-world applications. Furthermore, the decision to underclock the machine could have repercussions on the real-time application of the device.

For future research, in my opinion, it would be necessary to focus on ensuring a stable power supply, so the device can request and utilize all the necessary power. I also believe that, in addition to focusing on developing stable and accurate models, it is essential to conduct an

in-depth study and testing of optimized models that can run on low-power machines, aiming to minimize the consumption of computational resources, such as memory and processing power.

Appendix A

USB Camera

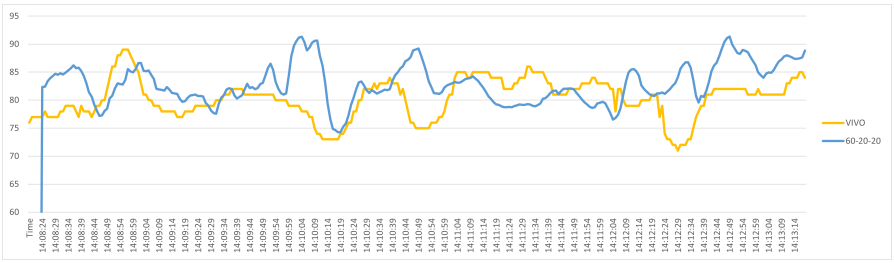


Figure A.1: PC results of forehead in natural light conditions

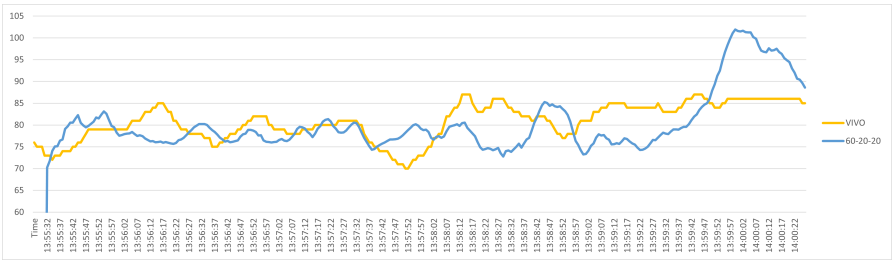


Figure A.2: Raspberry results of forehead in natural light conditions

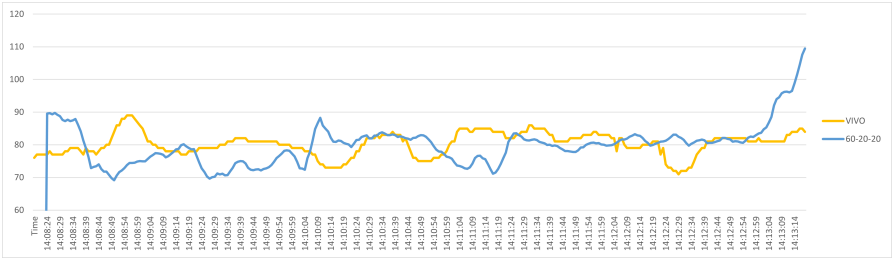


Figure A.3: PC results of left cheek in natural light conditions

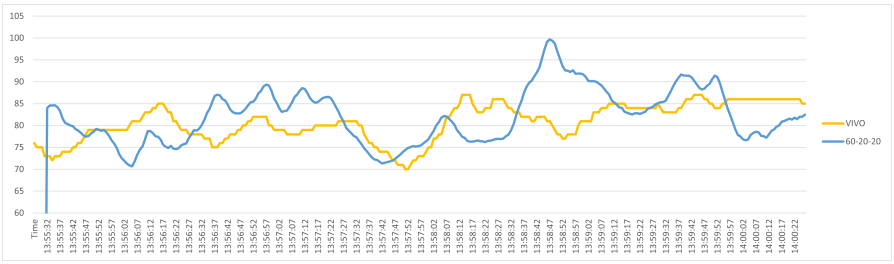


Figure A.4: Raspberry results of left cheek in natural light conditions

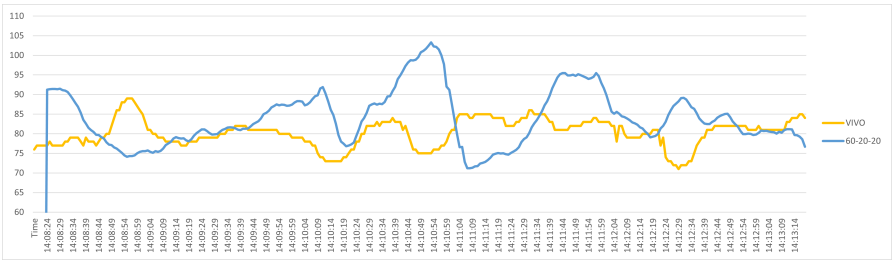


Figure A.5: PC results of right cheek in natural light conditions

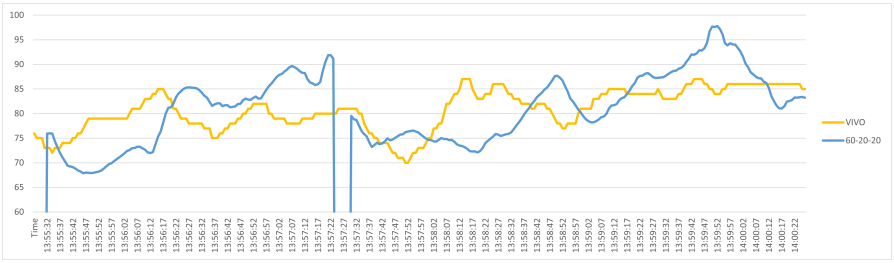


Figure A.6: Raspberry results of right cheek in natural light conditions

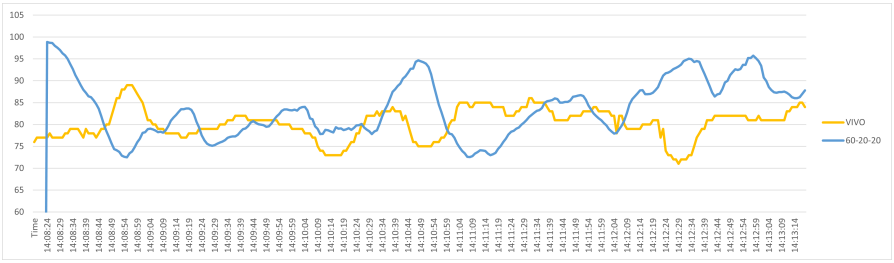


Figure A.7: PC results of total face in natural light conditions

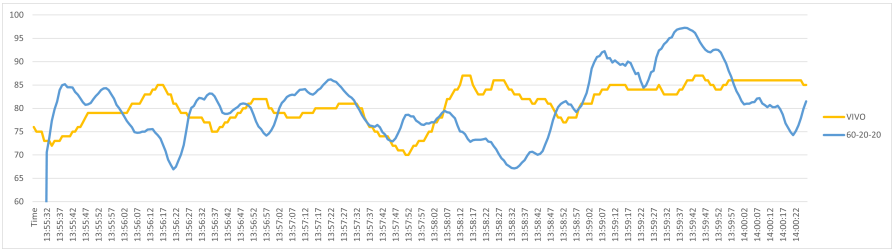


Figure A.8: Raspberry results of total in natural light conditions

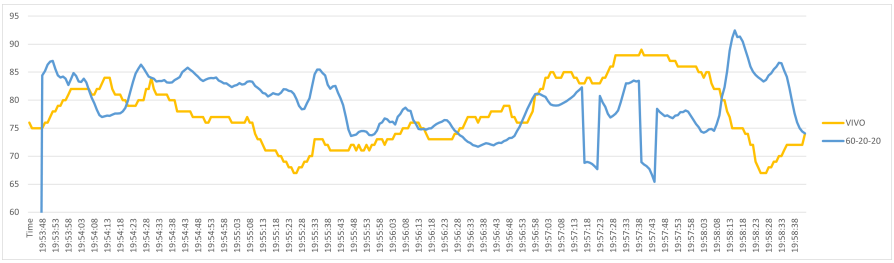


Figure A.9: PC results of forehead in direct artificial light conditions

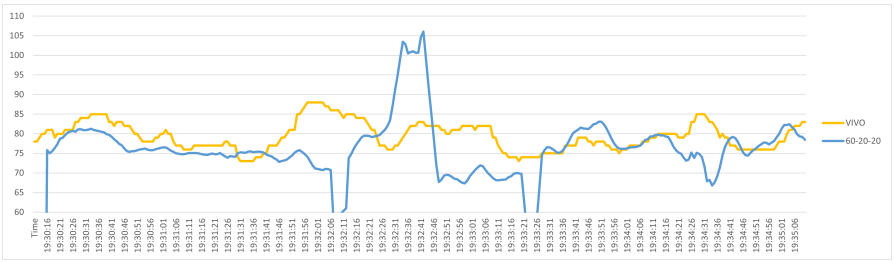


Figure A.10: Raspberry results of forehead in direct artificial light conditions

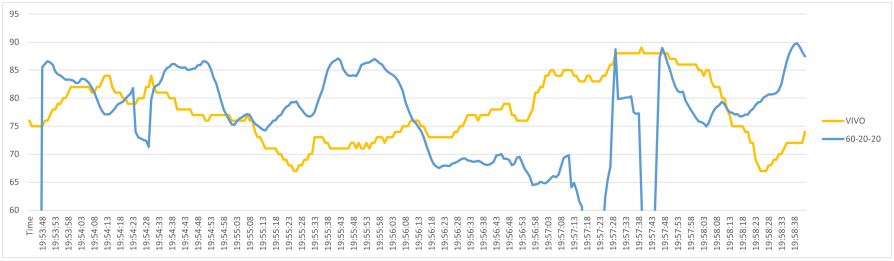


Figure A.11: PC results of left cheek in direct artificial light conditions

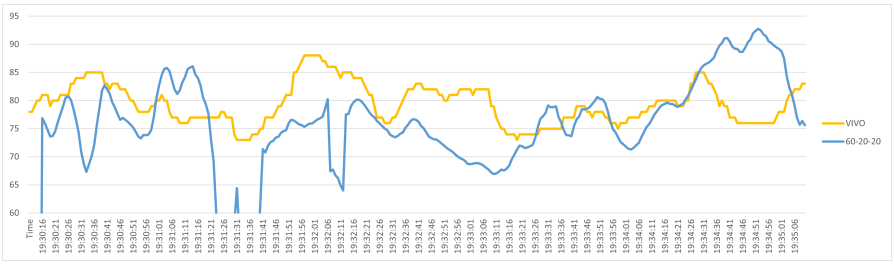


Figure A.12: Raspberry results of left cheek in direct artificial light conditions

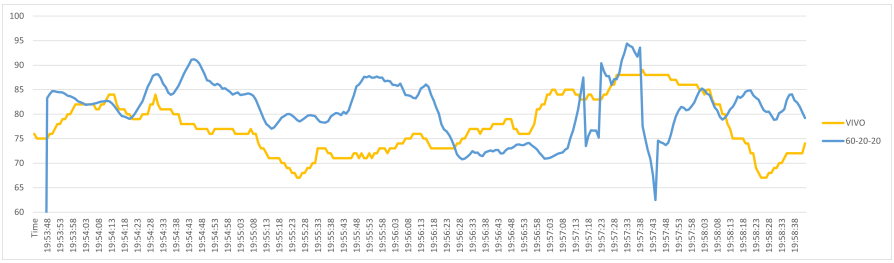


Figure A.13: PC results of right cheek in direct artificial light conditions

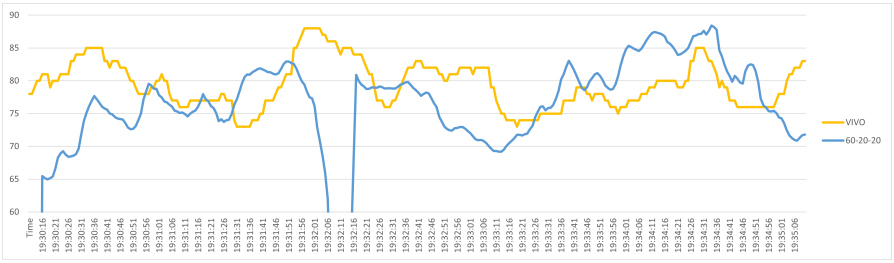


Figure A.14: Raspberry results of right cheek in direct artificial light conditions

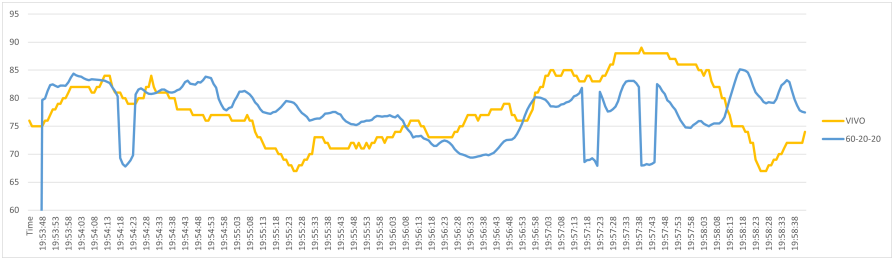


Figure A.15: PC results of total face in direct artificial light conditions

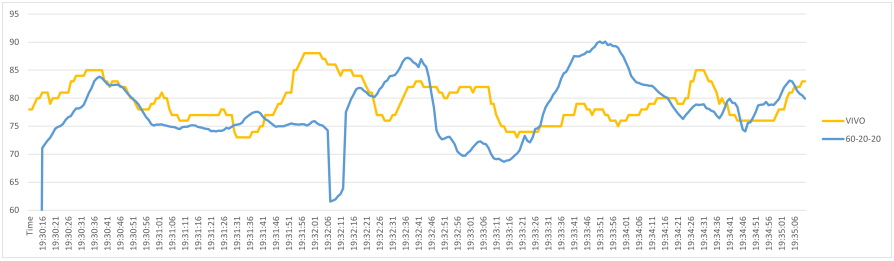


Figure A.16: Raspberry results of total face in direct artificial light conditions

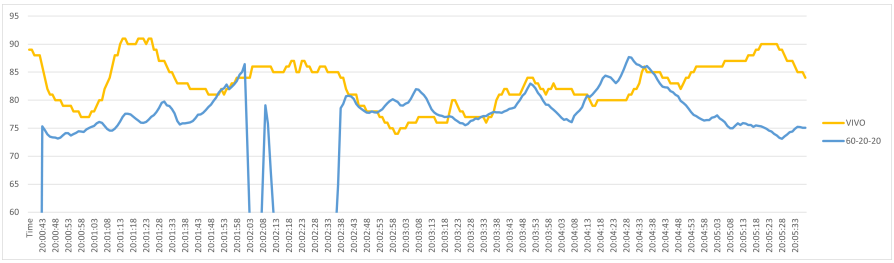


Figure A.17: PC results of forehead in indirect artificial light conditions

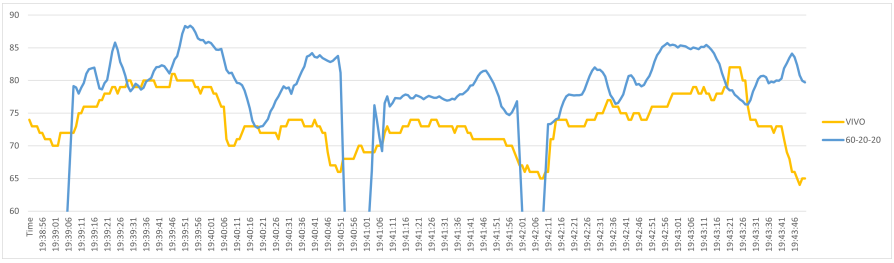


Figure A.18: Raspberry results of forehead in indirect artificial light conditions

Figure A.19: PC results of left cheek in indirect artificial light conditions

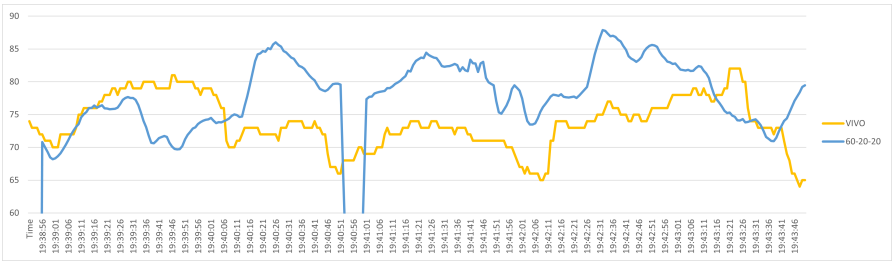


Figure A.20: Raspberry results of left cheek in indirect artificial light conditions

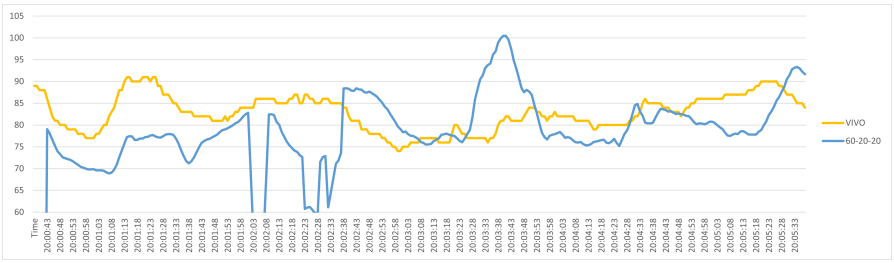


Figure A.21: PC results of right cheek in indirect artificial light conditions

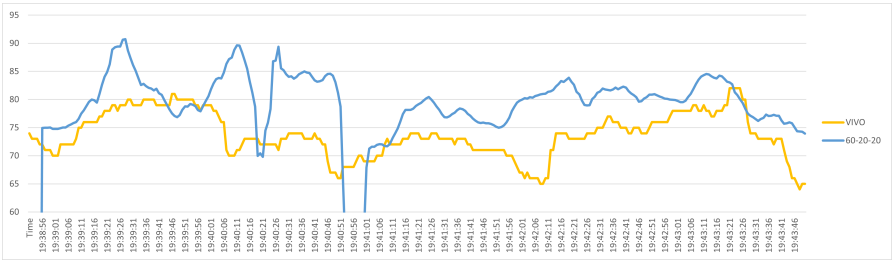


Figure A.22: Raspberry results of right cheek in indirect artificial light conditions

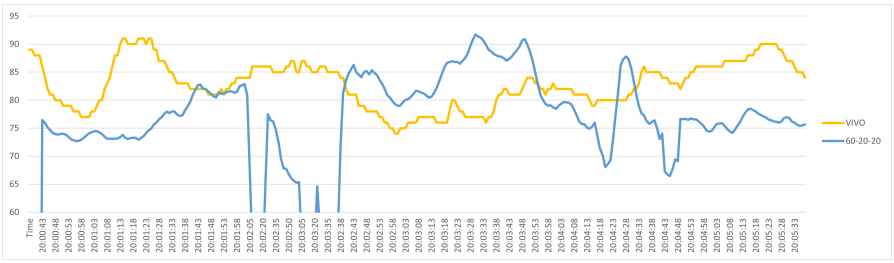


Figure A.23: PC results of total face in indirect artificial light conditions

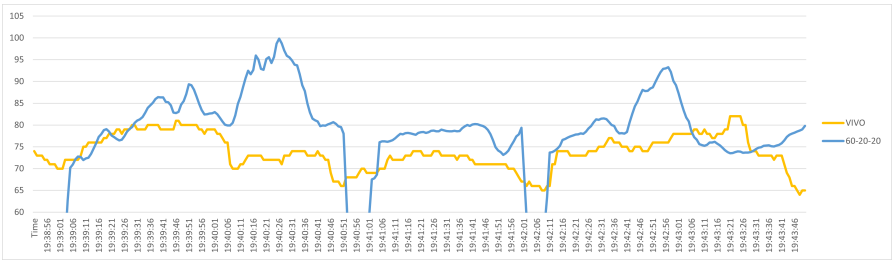


Figure A.24: Raspberry results of total face in indirect artificial light conditions

Appendix B

Raspberry Camera v3

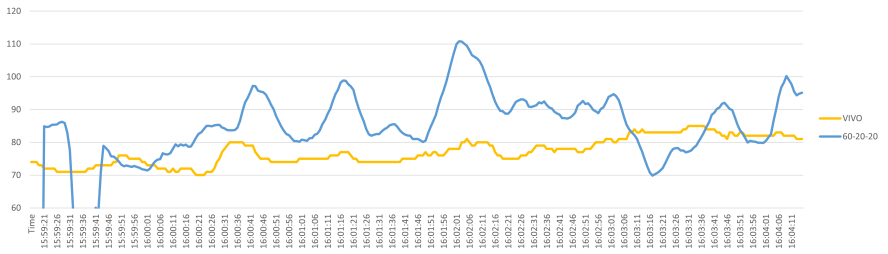


Figure B.1: Results of forehead in natural light conditions

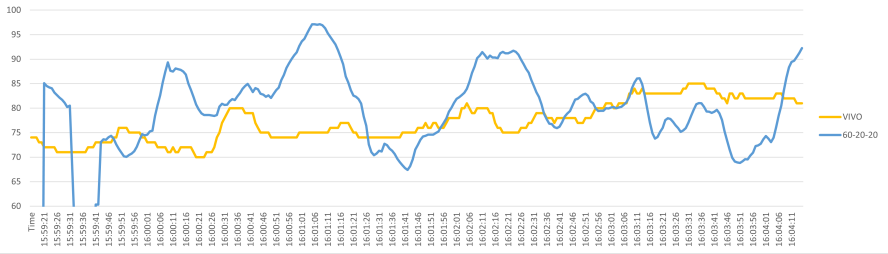


Figure B.2: Results of left cheek in natural light conditions

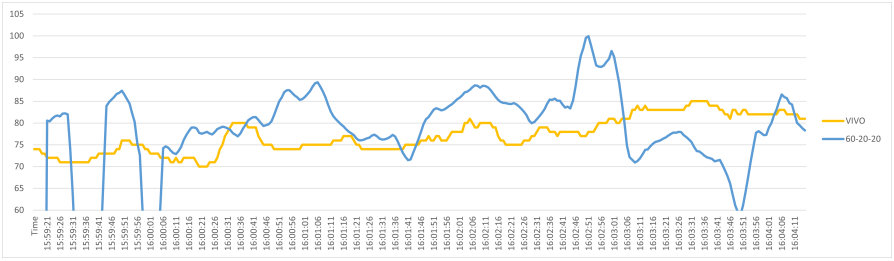


Figure B.3: Results of right cheek in natural light conditions

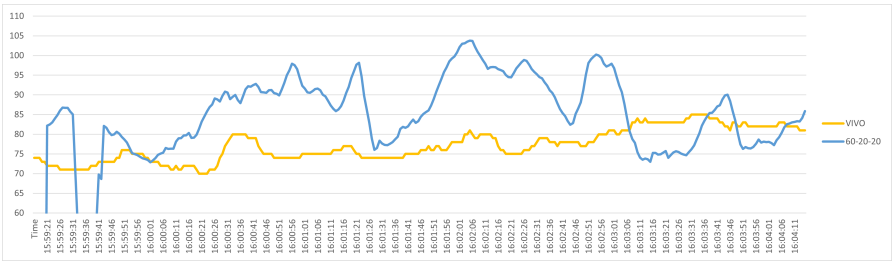


Figure B.4: Results of total face in natural light conditions

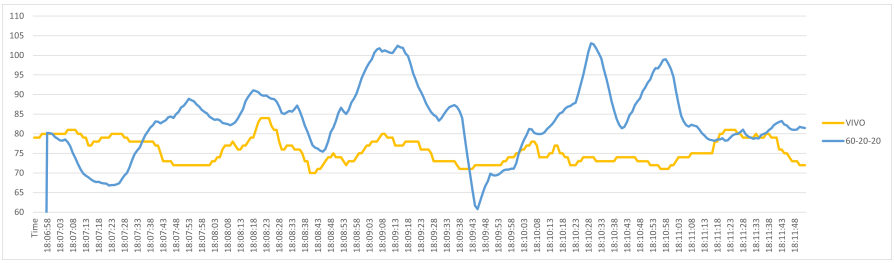


Figure B.5: Results of forehead in direct artificial light conditions

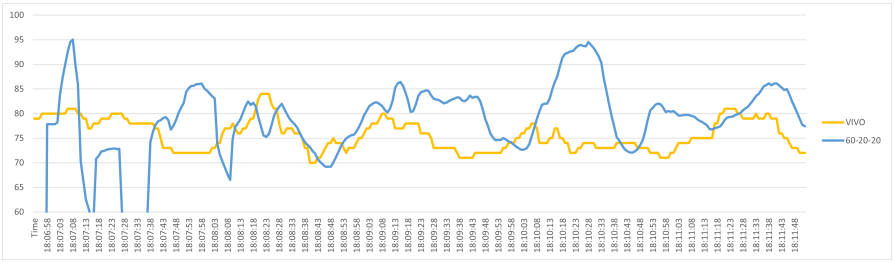


Figure B.6: Results of left cheek in direct artificial light conditions

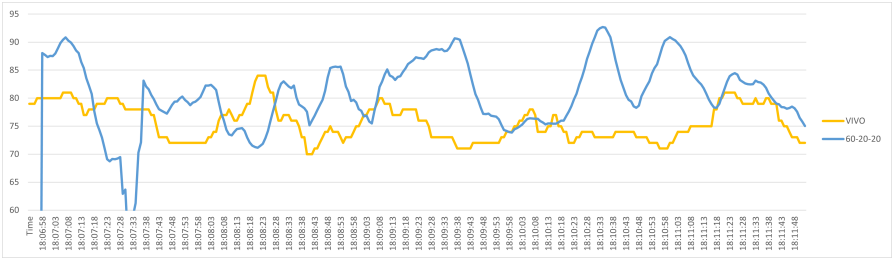


Figure B.7: Results of right cheek in direct artificial light conditions

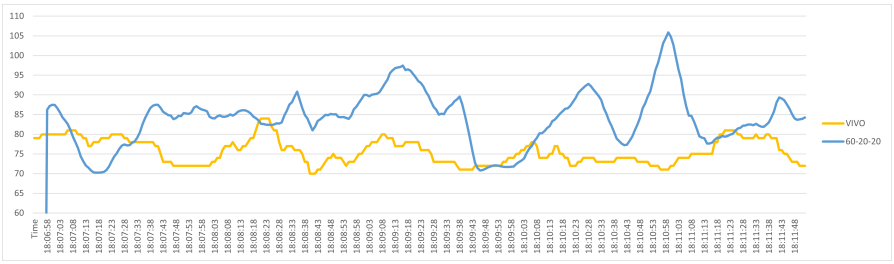


Figure B.8: Results of total face in direct artificial light conditions

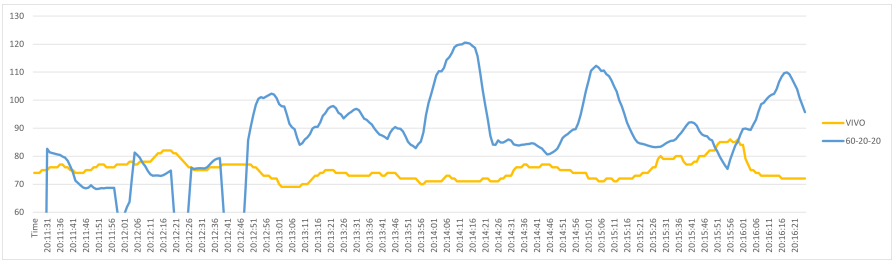


Figure B.9: Results of forehead in indirect artificial light conditions

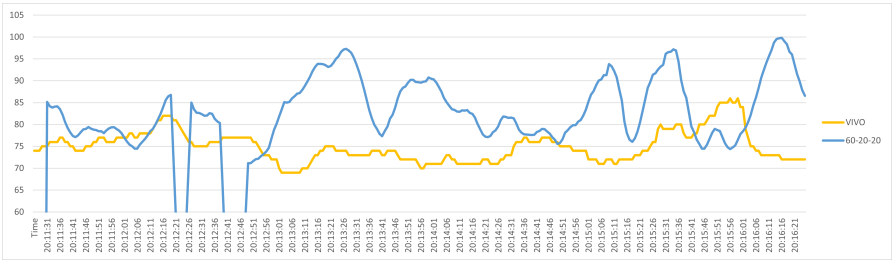


Figure B.10: Results of left cheek in indirect artificial light conditions

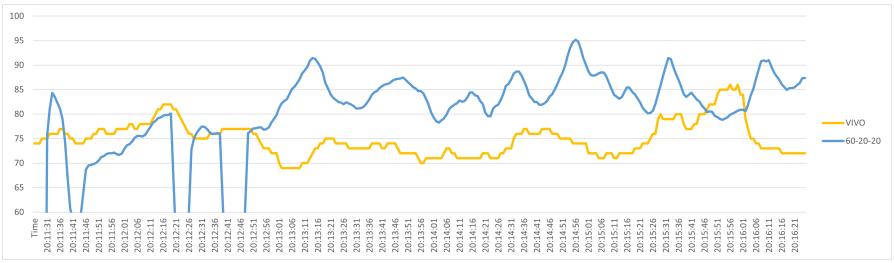


Figure B.11: Results of right cheek in indirect artificial light conditions

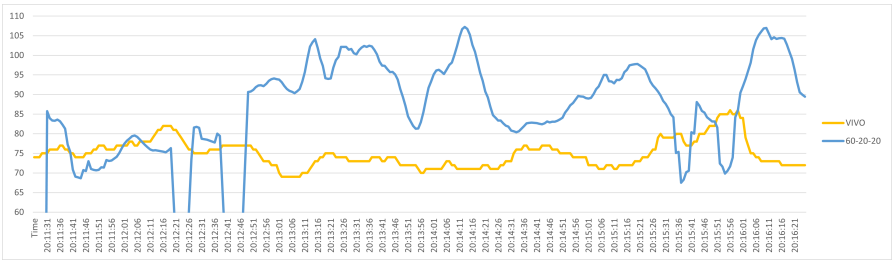


Figure B.12: Results of total face in indirect artificial light conditions

Appendix C

Raspberry Camera v3 Wide

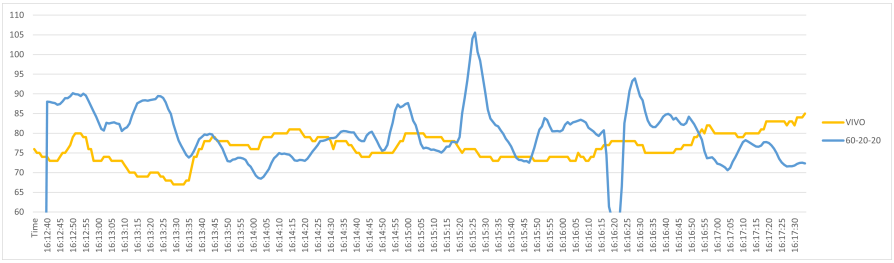


Figure C.1: Results of forehead in natural light conditions

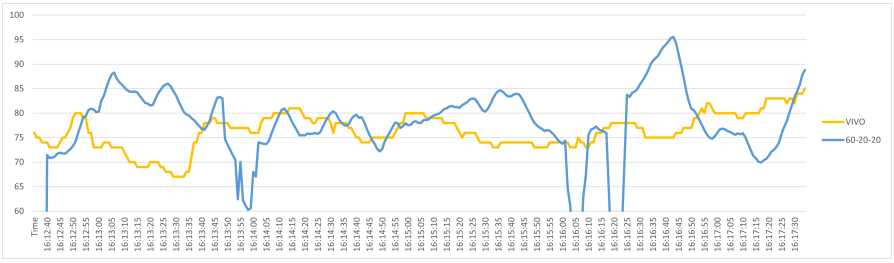


Figure C.2: Results of left cheek in natural light conditions

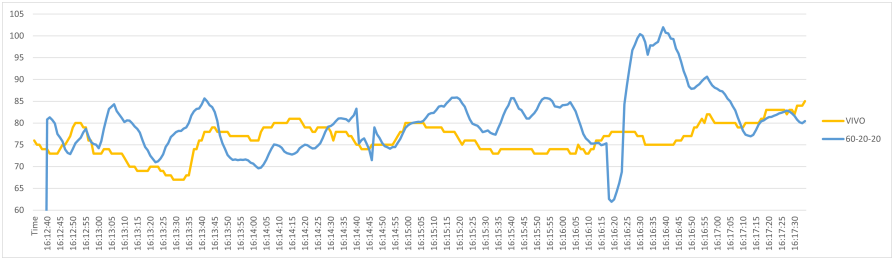


Figure C.3: Results of right cheek in natural light conditions

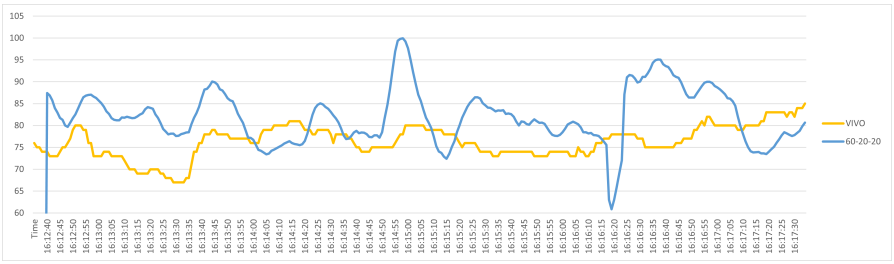


Figure C.4: Results of total face in natural light conditions

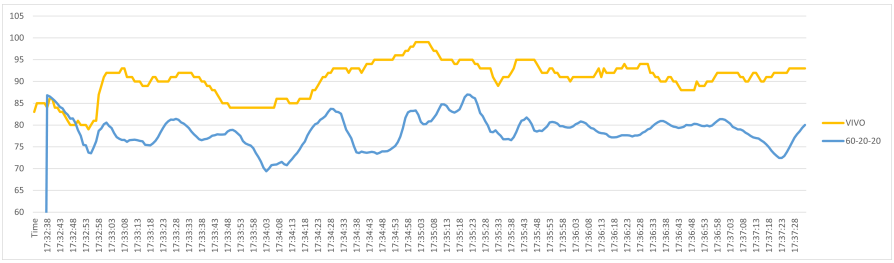


Figure C.5: Results of forehead in direct artificial light conditions

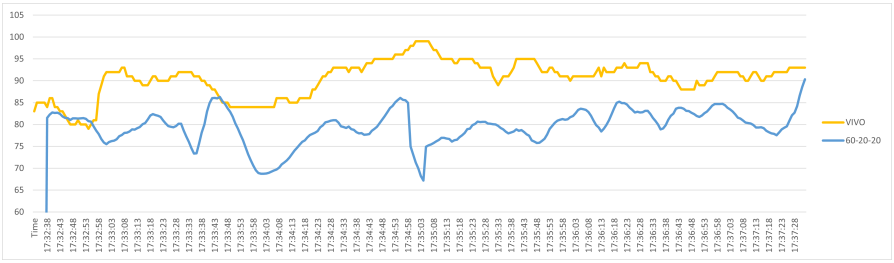


Figure C.6: Results of left cheek in direct artificial light conditions

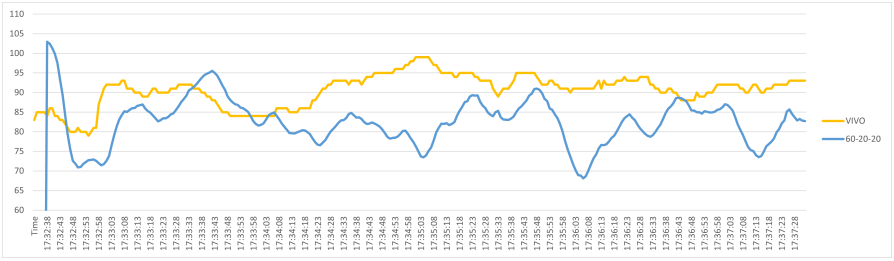


Figure C.7: Results of right cheek in direct artificial light conditions

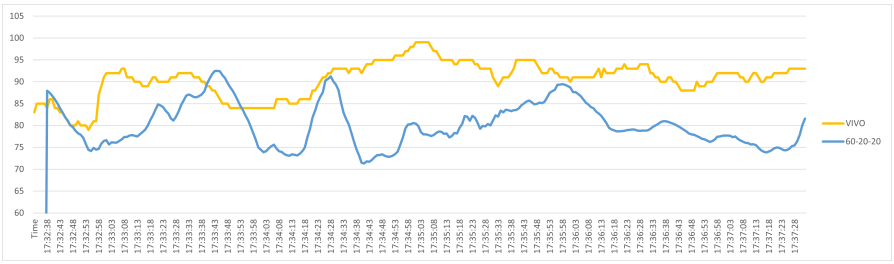


Figure C.8: Results of total face in direct artificial light conditions

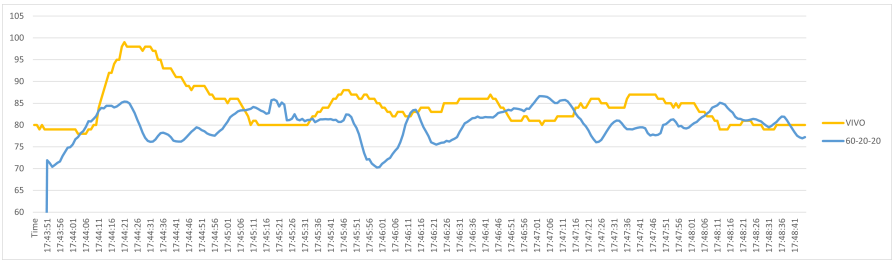


Figure C.9: Results of forehead in indirect artificial light conditions

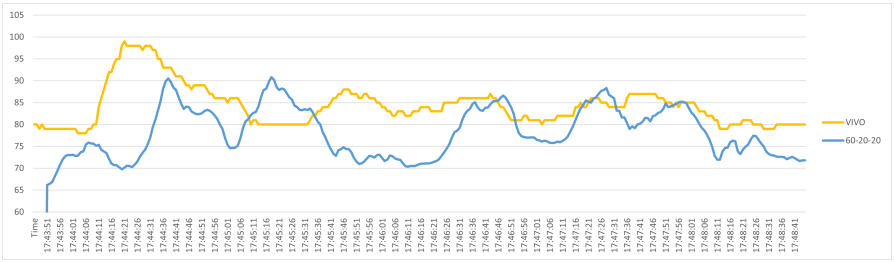


Figure C.10: Results of left cheek in indirect artificial light conditions

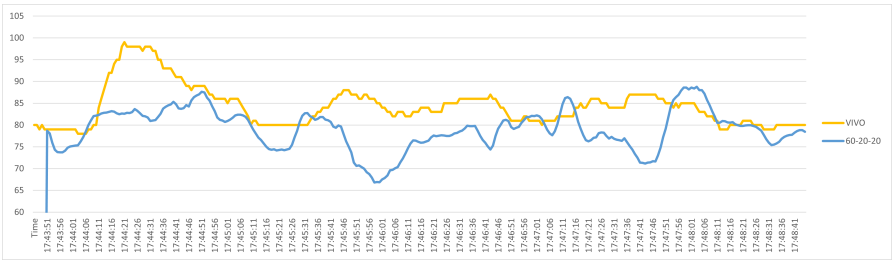


Figure C.11: Results of right cheek in indirect artificial light conditions

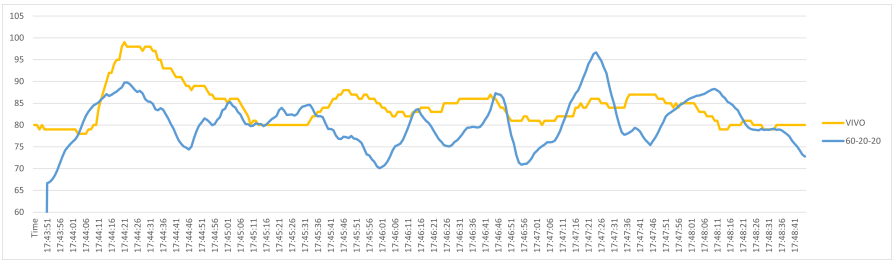


Figure C.12: Results of total face in indirect artificial light conditions

Appendix D

Raspberry HQ Camera

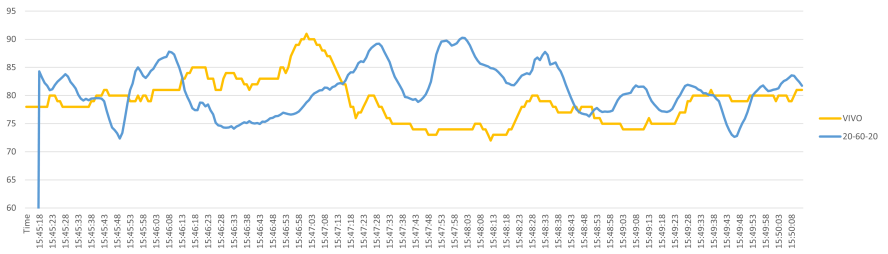


Figure D.1: Results of forehead in natural light conditions

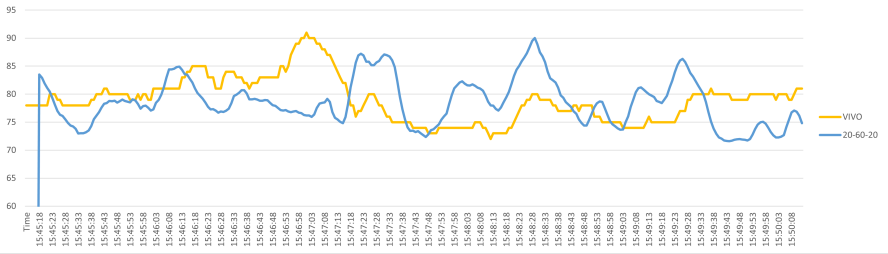


Figure D.2: Results of left cheek in natural light conditions

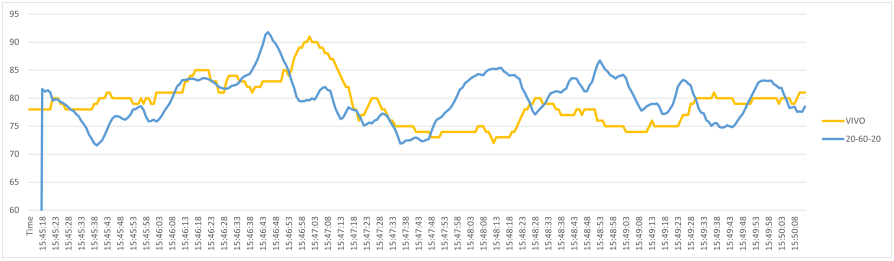


Figure D.3: Results of right cheek in natural light conditions

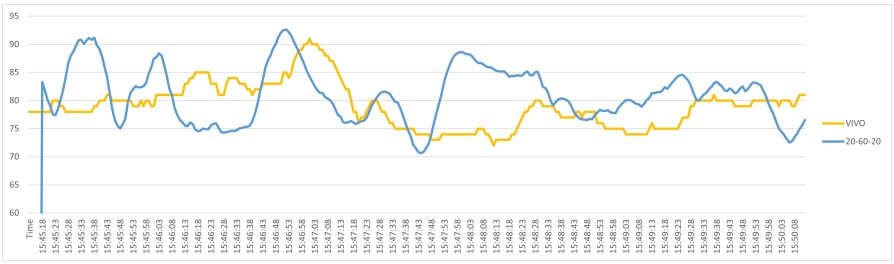


Figure D.4: Results of total face in natural light conditions

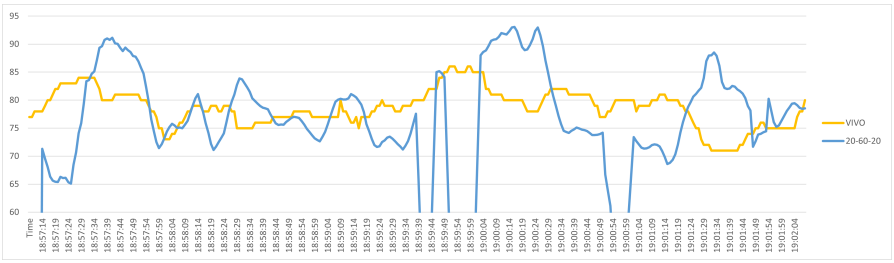


Figure D.5: Results of forehead in direct artificial light conditions

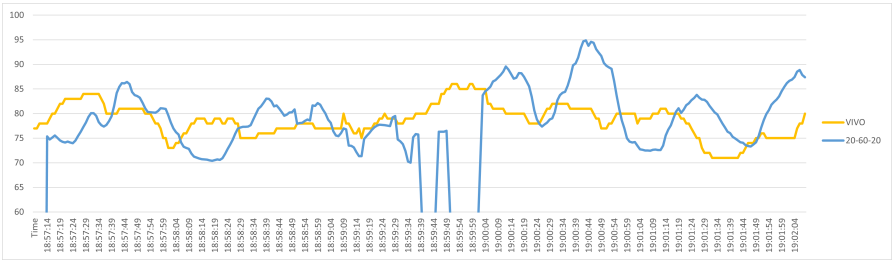


Figure D.6: Results of left cheek in direct artificial light conditions

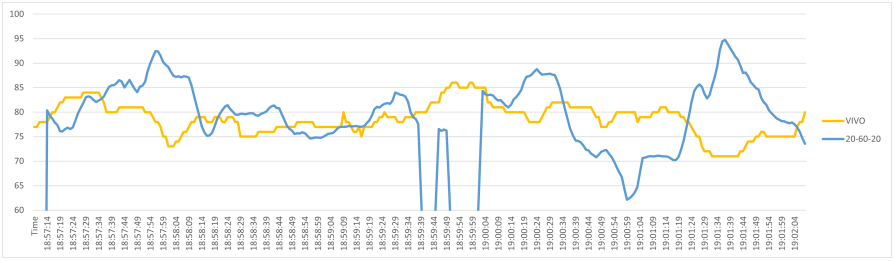


Figure D.7: Results of right cheek in direct artificial light conditions

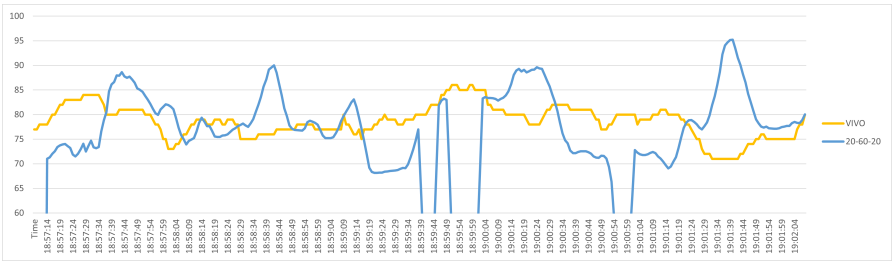


Figure D.8: Results of total face in direct artificial light conditions

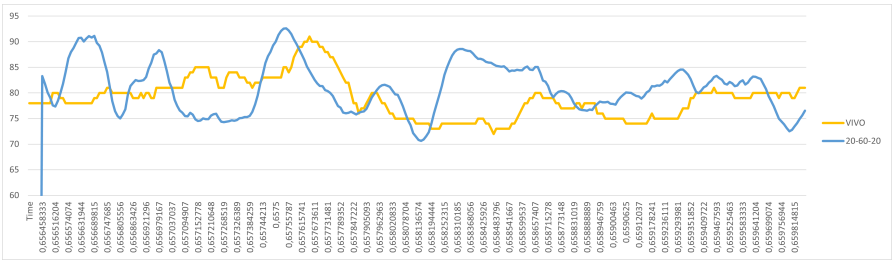


Figure D.9: Results of forehead in indirect artificial light conditions

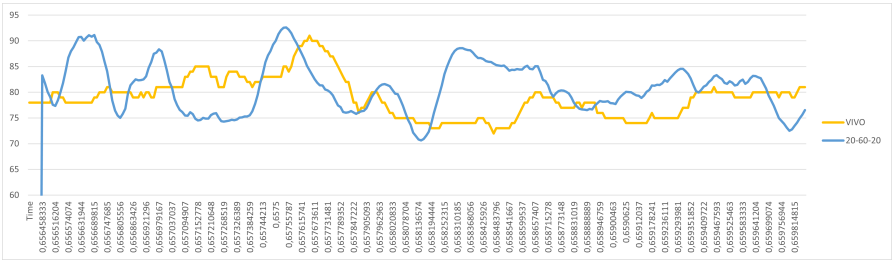


Figure D.10: Results of left cheek in indirect artificial light conditions

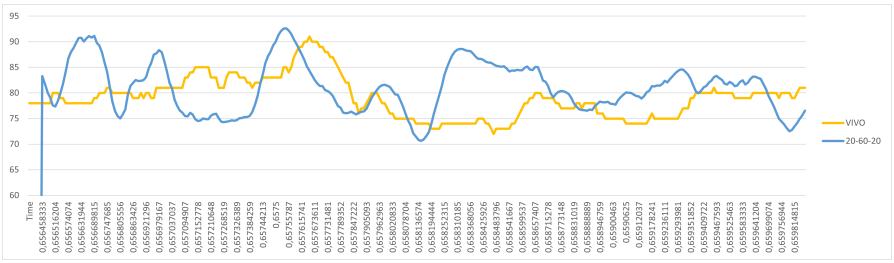


Figure D.11: Results of right cheek in indirect artificial light conditions

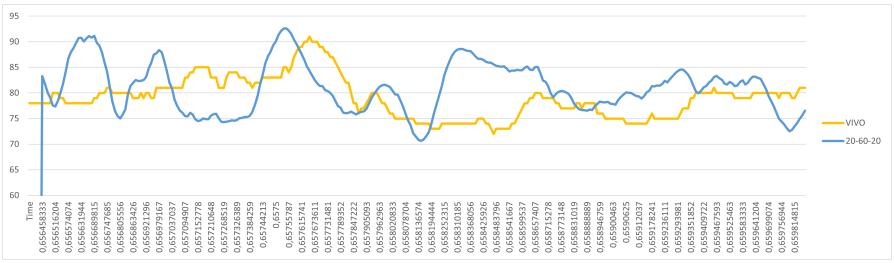


Figure D.12: Results of total face in indirect artificial light conditions

Bibliography

[1] Raspberry Pi Foundation. *Raspberry Pi 5 Product Brief*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf (cit. on pp. 3, 6).

[2] Raspberry Pi Foundation. *Camera Module 3 Product Brief*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/camera/camera-module-3-product-brief.pdf (cit. on pp. 6, 14).

[3] Raspberry Pi Foundation. *Camera Module 3 (Wide) Product Brief*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/camera/camera-module-3-product-brief.pdf (cit. on pp. 6, 16).

[4] Raspberry Pi Foundation. *Camera Module 3 (HQ) Product Brief*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/camera/camera-module-3-product-brief.pdf (cit. on pp. 6, 18).

[5] Raspberry Pi Foundation. *Raspberry Pi Active Cooler Product Brief*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/cooling/raspberry-pi-active-cooler-product-brief.pdf (cit. on p. 6).

[6] Arducam. *Arducam CS-Mount Lens for Raspberry Pi HQ Camera 8mm Focal Length with Manual Focus and Adjustable Aperture*. Accessed: 2025-03-09. 2025. URL: https://www.arducam.com/product/arducam-cs-mount-lens-for-raspberry-pi-hq-camera-8mm-focal-length-with-manual-focus-and-adjustable-aperture/ (cit. on p. 18).

[7] OpenCV Team. *OpenCV: Open Source Computer Vision Library*. Online. Available: https://docs.opencv.org/4.10.0/d1/dfb/intro.html. 2024 (cit. on p. 22).

[8] Anaconda. *Working with Conda Environments*. Accessed: March 6, 2025. 2024. URL: https://www.anaconda.com/docs/tools/working-with-conda/environments?utm_source=chatgpt.com (cit. on p. 22).

[9] Google AI. *MediaPipe Solutions Guide*. Accessed: 2025-03-06. 2024. URL: https://ai.google.dev/edge/mediapipe/solutions/guide?hl=it (cit. on p. 22).

[10] Satya Mallick. *Introduction to MediaPipe*. Accessed: 2025-03-06. 2023. URL: https://learnopencv.com/introduction-to-mediapipe/ (cit. on p. 22).

[11] HeartPy Team. *HeartPy: Python Toolkit for Heart Rate Analysis*. Accessed: 2025-03-06. 2024. URL: https://python-heart-rate-analysis-toolkit.readthedocs.io/ (cit. on p. 23).

[12] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You Only Look Once: Unified, Real-Time Object Detection». In: *arXiv preprint arXiv:1506.02640* (2015). URL: https://arxiv.org/abs/1506.02640 (cit. on p. 23).

[13] MediaPipe Documentation. *Getting Started - Troubleshooting*. Accessed: 2025-02-08. 2024. URL: https://mediapipe.readthedocs.io/en/latest/getting_started/troubleshooting.html (cit. on p. 23).

[14] Python Software Foundation. *Thread State and the Global Interpreter Lock*. Accessed: 2025-03-06. 2021. URL: https://docs.python.org/3.10/c-api/init.html?highlight=gil#thread-state-and-the-global-interpreter-lock (cit. on p. 23).

[15] Google. *Face Mesh with Mediapipe*. Accessed: 2025-03-06. 2024. URL: https://colab.research.google.com/github/spmallick/learnopencv/blob/master/Introduction-to-MediaPipe/MediaPipe-sample-solutions.ipynb (cit. on p. 27).

[16] Google AI. *MediaPipe Face Mesh*. Accessed: 2025-03-06. 2025. URL: https://github.com/google-ai-edge/mediapipe/blob/master/docs/solutions/face_mesh.md (cit. on p. 32).

[17] Google Developers. *MediaPipe 3D Face Transform*. Accessed: 2025-03-06. 2020. URL: https://developers.googleblog.com/en/mediapipe-3d-face-transform/ (cit. on p. 32).

[18] Patrik Hansen, Marianela García Lozano, Farzad Kamrani, and Joel Brynielsson. «Real-Time Estimation of Heart Rate in Situations Characterized by Dynamic Illumination using Remote Photoplethysmography». In: *Proceedings of the Conference on Remote Photoplethysmography*. FOI Swedish Defence Research Agency and KTH Royal Institute of Technology. 2023 (cit. on p. 33).

[19] Karan Chopra et al. «A Comprehensive Examination of Topographic Thickness of Skin in the Human Face». In: *Aesthetic Surgery Journal* 35.8 (2015), pp. 1007–1013 (cit. on p. 33).

[20] Linux Kernel Documentation. *Video4Linux2 (V4L2) API documentation*. Accessed: 2025-03-09. 2017. URL: https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/common.html (cit. on p. 49).

[21] Raspberry Pi Foundation. *Raspberry Pi Camera Guide*. Accessed: 2025-03-09. 2025. URL: https://datasheets.raspberrypi.com/camera/raspberry-pi-camera-guide.pdf (cit. on p. 49).

[22] SusanQQ. *UTKFace: A Large-Scale Dataset for Age, Gender, and Ethnicity Classification*. Accessed: 2025-03-29. 2021. URL: https://susanqq.github.io/UTKFace/ (cit. on pp. 52, 54).

[23] Gigi Reder Paolo Villaggio Anna Mazzamauro. *Fantozzi*. Accessed: 2024-03-30. 1975. URL: https://www.imdb.com/title/tt0071489/ (cit. on p. 56).

[24] Ultralytics. *YOLOv11 Documentation*. https://docs.ultralytics.com/it/models/yolo11/?utm_source=chatgpt.com#supported-tasks-and-modes. 2024 (cit. on p. 56).

[25] Ultralytics. *NCNN Integration and Installation Guide*. Accessed: 2024-03-30. 2024. URL: https://docs.ultralytics.com/it/integrations/ncnn/#installation (cit. on p. 57).

[26] Ultralytics. *YOLOv11 on Raspberry Pi*. Accessed: 2024-03-30. 2024. URL: https://docs.ultralytics.com/it/guides/raspberry-pi/#convert-model-to-ncnn-and-run-inference (cit. on p. 57).

[27] Rodrigo Castellano Ontiveros, Mohamed Elgendi, Giuseppe Missale, and Carlo Menon. «Evaluating RGB channels in remote photoplethysmography: a comparative study with contact-based PPG». In: *Frontiers in Physiology* 14 (2023). ISSN: 1664-042X. DOI: 10.3389/fphys.2023.1296277. URL: https://www.frontiersin.org/journals/physiology/articles/10.3389/fphys.2023.1296277 (cit. on p. 60).