Tampere University

Tero Vierimaa

AN AUTOMATIC CALIBRATION SYSTEM SOFTWARE FOR A MEDICAL LASER DEVICE

Master of Science Thesis
Faculty of Information Technology and Communication Sciences
February 2022

ABSTRACT

Tero Vierimaa: An Automatic Calibration System Software For a Medical Laser Device
Master of Science Thesis
Tampere University
Embedded Systems
February 2022

Integrating semiconductor lasers in a variety of different applications such as consumer electronics, manufacturing processes, aviation and even in medical applications makes them very ubiquitous in today's society. Further progress in the development of semiconductors and its advanced applications have generated new emerging markets. For instance, complex laser systems for the treatment of tumors and infections are recognized as a new market.

This thesis introduces an advanced automatic calibration software system to be integrated in the production process of a medical laser device. The medical device includes two lasers: a treatment beam and an aiming beam, adding to the calibration software's complexity. Additionally, the laser system consists of two subsystems: a laser device and an optical beam shaper unit. Once constructed, the calibration system would enable the medical device to be characterized and calibrated fully to ensure that it operates under its specification.

The aim of this project was to provide the company with a working solution for the production environment that passes all regulatory and internal requirements. The development process began with defining a list of specifications for the calibration system and its software, which was followed by gradually integrating the software with the rest of the calibration system. The inclusion of the theoretical background presented in this thesis is necessary to understand the challenges and corresponding technical decisions that were being made during the course of this project.

The backbone of the software architecture and its core mechanisms are illustrated as an overview of the automatic calibration system. In addition, the implemented components and algorithms, including the necessary performance optimizations for selected subsystems, are described. Some additional constraints, that needed to be taken into consideration during the development of the calibration software, included software licensing and the system's safety.

After the implementation the calibration system was validated and verified to ensure that it meets its specifications and performance requirements. To improve the calibration system a few ideas are presented, such as optimizing the beam profiling procedure. Ultimately, the calibration system as well as its software were concluded to be suitable and capable of being utilized in the production process of the medical laser device.

Keywords: characterization, calibration, machine vision, laser, software

TIIVISTELMÄ

Tero Vierimaa: An Automatic Calibration System Software For a Medical Laser Device
Master of Science Thesis
Tampereen yliopisto
Sulautetut järjestelmät
Helmikuu 2022

Puolijohdelasereiden integrointi lukuisiin sovelluksiin, kuten kuluttajaeletrokniikkaan, valmistusprosesseihin, ilmailuun ja jopa lääkinnällisiin sovelluksiin tekee niistä hyvin käytettyjä nyky-yhteiskunnassa. Viimeisin edistys puolijohteiden kehityksessä ja niiden haastavissa sovelluksissa on lisäksi luonut näille uusia markkinoita. Esimerkikkinä tällaisesta ovat monimutkaiset laserjärjestelmät lääkinnälliseen käyttöön, kuten kasvainten ja infektioiden hoitoon.

Tässä työssä esitellään automaattinen ohjelmisto osana kalibrointijärjestelmää, joka tullaan integroimaan osaksi lääkkinnällisen laserlaitteen valmistusprosessia. Ohjelmistoa monimutkaistaa lääkinnällisen laitteen kaksi laseria: hoitosäde sekä tähtäyssäde, jotka vaativat erilliset kalibroinnit. Lisäksi laite sisältää kaksi alijärjestelmää: laserlaitteen sekä optisen säteenmuokkainyksikön. Valmistuttuaan kalibrointijärjestelmä mahdollistaisi lääkinnällisen laitteen kattavan kalibroinnin ja karatkerisoinnin, mikä takaisi laitteen toiminnan spesifikaationsa mukaisesti.

Tämän projektin tavoitteena oli tuottaa yritykselle toimiva ratkaisu tuotantoympäristöön, joka läpäisee kaikki viranomais- sekä sisäiset vaatimukset. Kehitysprosessi alkoi määrittelemällä vaatimukset kalibrointijärjestelmälle ja sen ohjelmistolle, mitä seurasi koko järjestelmän asteittainen integrointi. Teoreettisen osuuden sisällyttäminen tähän työhön on välttämätöntä teknisten haasteiden ja ratkaisuden ymärtämiseksi ja perustelemiseksi.

Ohjelmiston arkkitehtuurin perusidea sekä sen keskeiset mekanismit ovat esitettynä osana työtä. Lisäksi osa toteutetuista komponenteista sekä algoritmeista on kuvattuna, mukaanlukien välttämämttömät suorituskykyoptimoinnit, joita projektin aikana tehtiin. Muutamat lisärajoitteet, jotka piti huomioida kalibrointiohjelmistoa tehtäessä, sisälsivät esimerkiksi ohjelmistolisenssit sekä järjestelmän turvallisuuden.

Kun kalibrointijärjestelmä oli toteutettu, varmistettiin spesifikaation ja suorituskykyvaatimusten täyttyminen validoimalla ja verifioimalla järjestelmä. Järjestelmän lisäkehittämiseksi on esitelty muutamia jatkoideoita, kuten esimerkiksi säteenprofilointiproseduurin suorituskykyoptimointi. Lopulta kalibrointijärjestelmä ja sen ohjelmisto todettiin soveltuvaksi ja kykeneväksi käytettäväksi lääkinnällisen laserjärjestelmän tuotantoympäristössä.

Avainsanat: karakterisointi, kalibrointi, konenäkö, laser, ohjelmisto

PREFACE

The project was built during early 2019 to late 2021. Writing of this thesis was done during 2021 and early 2022. During this time, I had the chance to deeply research and study the subject and implement a software for characterizing a medical laser device.

Firstly I would also like to thank my supervisors, Professor Karri Palovuori and University Lecturer Erja Sipilä for their help and support throughout the writing process. Secondly, I would also like to thank CEO Seppo Orsila and Dr. Petteri Uusimaa for the opportunity to work on such an interesting and challenging project. I also want to thank Visa Kaivosoja, Juha Lemmetti, Jukka-Pekka Alanko, for their guidance during the project. Additionally, I want to thank Matius Hurskainen, Ivan Baldin, Vilma Motturi and Lasse Linkola for their expertise and help during the project.

Thirdly, thank you to my family and friends for all the help you gave me before and while I was writing this thesis. Lastly, thank you Laura for your help and fullest support during the project.

Tampere, 9th February 2022

Tero Vierimaa

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF PROGRAMS AND ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

ADC	Analog-Digital Converter
AsyncIO	A Python core library for writing concurrent code
CB	Certified Body
CPU	Central Processing Unit
DAC	Digital-Analog Converter
DSP	Digital Signal Processing
FDA	Food and Drug Administration
FIR	Finite Impulse Response
FPS	Frames Per Second
GigE	Gigabit Ethernet
GIL	Python Global Interpreter Lock
GPIO	General Purpose Input Output
GPL	General Public License
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IEC	International Electrotechnical Comission
IECEE	IEC System of Conformity Assessment Schemes for Electrotechnical Equipment and Components
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
LIDAR	Light Imaging, Detection and Ranging
LIV	Laser / Light Current Voltage Curve
MVC	Model-View-Controller -paradigm
NumPy	Python package for numerical computing
OOP	Object-Oriented Programming
OpenCV	Open source computer vision library
OS	Operating System
PDT	Photodynamic Therapy
PMA	Premarket Approval

PMN	Premarket Notification
Python	Python programming language
SPI	Serial Peripheral Interface
TAU	Tampere University
TCP/IP	Transmission Control Protocol/Internet Protocol
TUNI	Tampere Universities

1 INTRODUCTION

For the past few decades the rapid emergence of new applications for lasers has been significant. Unlike the initially large and expensives ones, a modern day semiconductor laser is small and has high efficiency. Throughout the past decades the industry and market has been steadily growing [1]. In USA alone the laser market's size increased 250% between years 2009 and 2017 [2]. One of the submarket for laser systems is the use of lasers in medical applications. In addition to treating cancer and tumors with photodynamic therapy, potential use cases include treating psoriasis, acne and ultraviolet-damaged skin [3], [4]. In many of the medical applications, the aim is to deliver a highly selective drug to the correct location in the patient's body, and release or activate the drug with the assistance of laser light. This makes targeted, precise treatment for various diseases possible. Photocynamic therapy (PDT) can be used, but is not limited, for treating infections and various cancers and tumors, such as a breast cancer [5], [6], [7].

Traditional industrial laser applications include, for instance, barcode scanners, laser cutting, telecommunications and digital storage systems. New applications include LIDARs (Light imaging, detection and ranging) and various manufacturing processes. In modern applications various technologies are converged, ranging from machine learning and biomedical engineering to optical systems. For instance, with LIDARs, the development of computing power and new signal processing techniques has allowed more sophisticated methods for processing sensor data [8].

In this thesis, a software for a medical laser device's automated calibration is designed and implemented. The aim is to have a system that can be used in a production environment for calibration a medical laser device. Buildling such system and using it for production imposes regulatory challenges, since the laser system is used for a medical application. Additionally, there are various internal and external interests directed towards the calibration system. An overall architecture and selected technical decisions and implementations are described.

Although the thesis focuses solely on the software implementation, it incorporates multiple physical devices and communication with them. Technologies involved include electronics, optics, signal processing, machine vision and databases. For calibrating the medical system, a multitude of various characterizations is required. Finally, a verification and validation protocol against the requirements and specifications are concluded. A key output for the project work is for it to pass the verification and

validation protocol. Only after a successful validation may it be used for a production of the medical laser device.

The first chapter focuses on background factors needed for understanding the technical challenges and decisions being made during the project work. The second chapter lays out a simplified specification for the laser device and the calibration system used for calibrating the medical laser device. Chapter three focuses on the implemetation of the calibration software. Since the implemented software is of a fairly large magnitude, only a subset of the implemented funtionalities are presented in this thesis. These selected implemenation details are presented in chapter five. In chapter four a complete validation for the software is presented. The purpose of the validation process is to make sure that the software is indeed suitable for the use that it has been specified for. Finally, chapter five introduces a few possibilities for improving the implemented software.

2 BACKGROUND

In this chapter, the background for the development task process is described. First a short summary of a laser's key characteristics are presented. Then a technical background for the selected technologies are covered. The background focuses more on the relevant theory needed for constructing the software for the calibration system than electrical or optical characteristics. Although the programming language, any framework or library that has been used in this project is not a focus of this thesis, they are central for understanding the technical decisions that have been made during the project.

2.1 Laser diode characteristics

A semiconductor laser is a device that is widely used in many fields, such as telecommunications, manufacturing, chemistry and medical systems. To briefly explain, a laser semiconductor operates in a mode known as stimulated emission. In such a mode, laser is emitting light in an extremely narrow bandwidth in a coherent fashion. Laser emission is not completely monochromatic, but the central wavelength of ±1nm contains roughly 99% of the optical power. If not properly lasing, the laser diode operates as a normal light emitting diode, emitting a wider bandwidth. A laser's wavelength can vary anywhere from infrared to ultraviolet. Between these modes, in the lasing thershold area, the emission is something in between. [9]

A key figure for electrically controlling laser diodes is Light Current Voltage (LIV) curve [10]. An example of such a curve is shown in Figure 2.1. The exact curve is always unique to each module, while its overall limits are determined by the structure of the laser chip. It is easy to identify important parameters in reference to electrical and thermal control from an LIV curve. The LIV curve describes the relation between the current, the voltage and the output energy or power from the laser diode.

When examining Figure 2.1, three distinct operating areas can be identified: the non lasing-area in the lowest current under 550 mA, a threshold area around 550-600 mA and finally, the linear area in currents over 600 mA. Between these three areas, the non-lasing area is the least noteworthy, since the laser is effectively operating as a light emitting diode (LED) and a stimulated emission does not occur. In the threshold area the laser slowly begins lasing and current-to-output-power ratio is nonlinear. After the threshold area, the power is linear relative to the current passing through the laser module. For

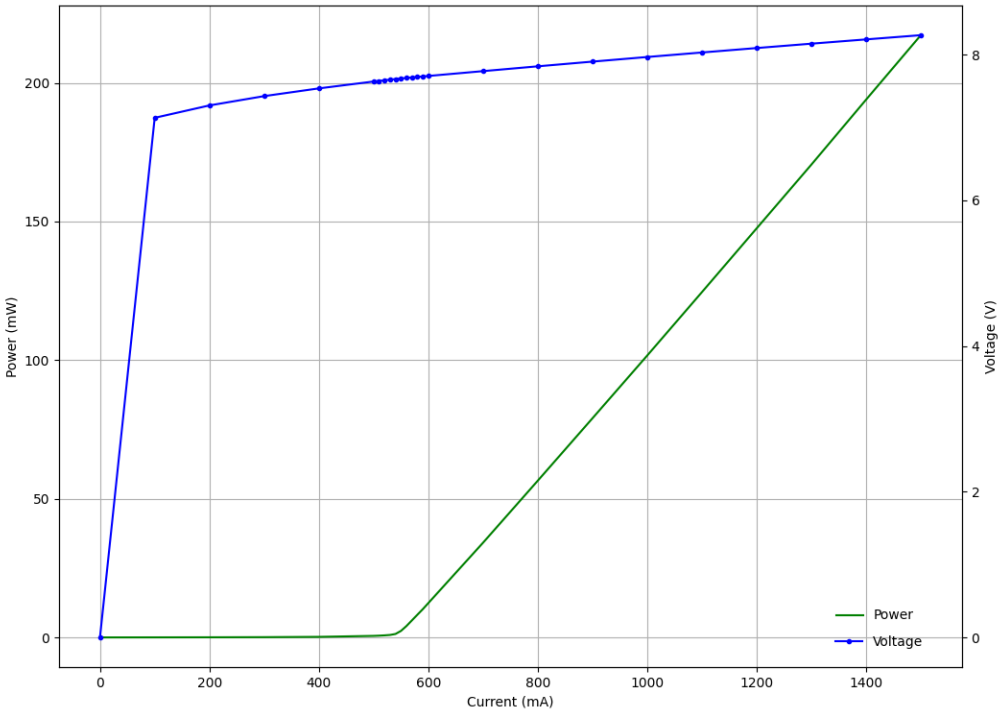

Figure 2.1. *Laser module LIV characteristics.*

successfully driving a laser module with an accurate output power, each of the three areas need to be handled separately in the laser's control logic.

In addition to the LIV, the effect of the temperature in respect to the output power is shown in Figure 2.1. The temperature does not change the overall characteristic of the LIV, meaning that each of the three areas are still present and in the same order. Instead, the laser chip's temperature shifts both the threshold area and the linear area to the lower or higher current. In other words, the laser chip's output power depends on both the current and the module's temperature. Hence, the temperature must be monitored and actively controlled to stabilize the output wavelength.

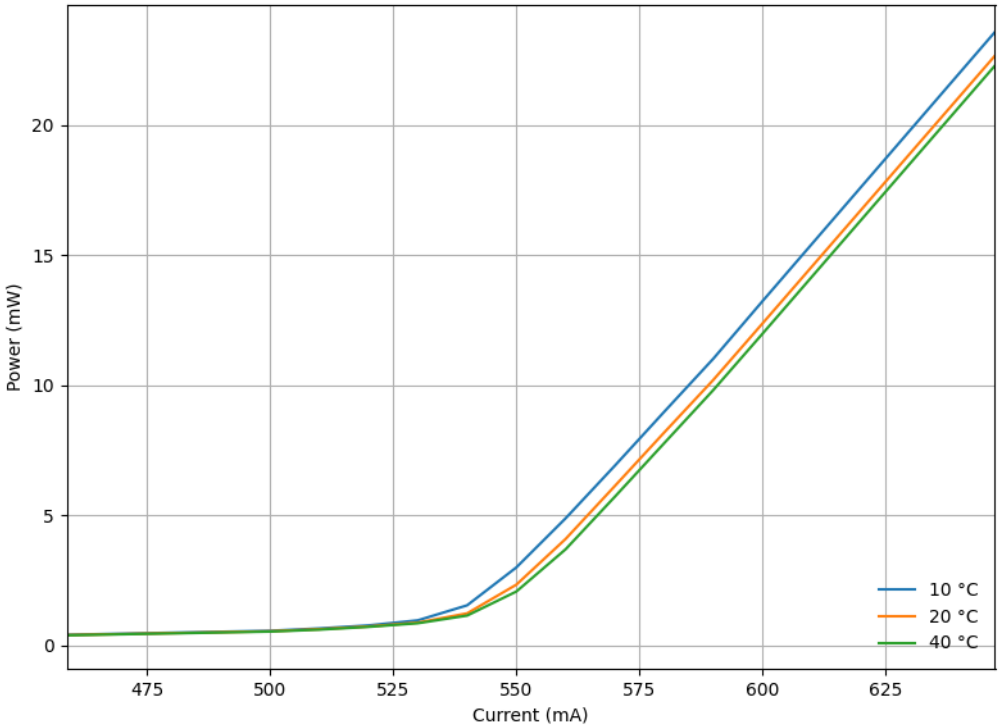

Figure 2.2. *Laser module LIV temperature effect*

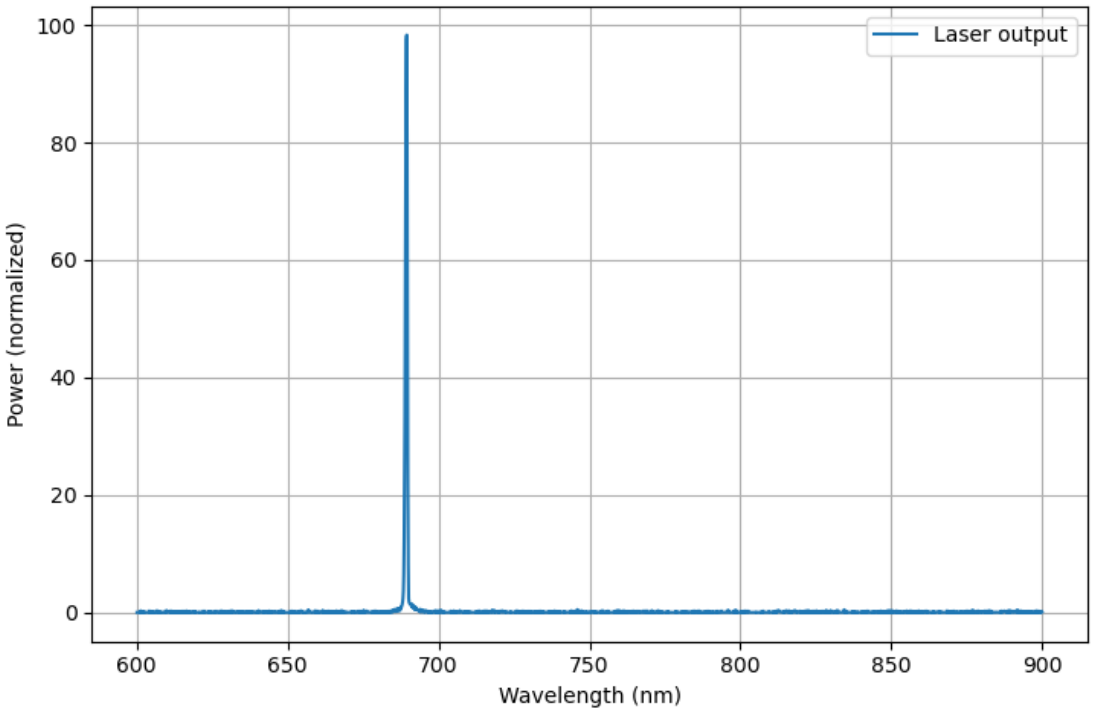

Figure 2.3. *Sample laser spectral characteristics*

When properly lasing, a laser has an extremely narrow spectral width, which is a key characteristic for a laser. The laser's exact central wavelength varies between different modules. An example of a laser's spectral characteristics is shown in Figure 2.3.

Depending on application the exact wavelength and power may be critical. In such applications it is necessary to monitor the laser diode's temperature to mitigate such faulty states. As seen in Figure 2.2 the module's temperature affects the output power. Usually a laser chip requires active cooling and possibly heating to stabilize high power operation. If the temperature drifts too far from specified, the laser system is disabled altogether to protect the system and the application.

By default, laser diodes are built as single chips. Depending on the application and required output power, a single laser module is constructed by combining multiple laser chips into one enclosure. A single module, approximately 1-5 cm in size, can contain a dozen of laser chips. An optical output is combined and directed to one or more fibers and usually the diodes are electrically connected in series. Additionally a module may contain a thermal sensor and a photodiode for monitoring the output power.

2.2 Medical device regulation

According to the US Food and Drug Administration (FDA), a medical device is defined as any device that is designed to be used for diagnostics, or altering the anatomy of the human body, excluding purely chemical mechanisms [11]. Further classification is applied in order to mitigate risks for patients and other parties.

The FDA categorizes medical devices into three main categories by the risks they pose and their assured effectiveness [12]. These classes are: I for lowest risk, II for moderate risk and III for high health risk [11]. Laser devices generally go directly to class III, since their risks are categorized as severe. In addition, most new devices which do not have any applicable predicate products must automatically pass the Pre Market Notification (PMA) process [13]. In order to pass class III regulations, the device must pass the PMA. This requires a significant amount of resources and time from any company. For classes I and II, the medical device can be granted approval, when it is compatible with another device that already has an approval, it is sometimes sufficient to prove its similarity to this existing device [14]. This so called Premarket Notification or a 501k is a much lighter alternative to PMA. PMA stresses the demonstration that the new device is in fact equivalent to the existing approved device both in terms of safety, their operation and effectiveness. For instance, a PMA requires a clinical study in addition to fulfilling the class I and class II clinical evidence [13]. 501k, on the other hand, aims to indisputably prove that the new device is, in fact, comparable to the original device.

The regulatory documentation, which aims to demonstrate the safety and validity of a medical device, is important for any company seeking to bring their product to the market. Furthermore, the exact requirements vary between countries, increasing the amount of work by multitude. To overcome this workload, an international system for mutual acceptance has been established by International Electrotechnical Comission (IEC). The system, known as IEC System for Conformity (IECEE) recognizes 3rd party actors known as Certification Bodies (CB) CBs are then delegated to do the concrete

verification and testing for products. After a successful test report and certification from a CB, a manufacturer can apply for approval of the device in the particular country or area.

2.3 Linux operating system

Linux, a free and open-source operating system is licensed with GNU General Public License (GPL). Being open-source makes it possible for anyone to modify and develop the operating system to better suit their needs without a fear of vendor lock-in.

This has lead to a large community of developers, which has enabled a healthy ecosystem around the operating system. Today, Linux is found in virtually any area: mobile phones, medical devices, industrial systems and network routers. The reason for this widespread adoption is, simply put, because Linux provides excellent functionality and flexibility for various environments and devices, and extending it is made easy.

Although Linux is designed, as per Unix philosophy, as a general purpose operating system ranging from desktops to servers, it is possible to configure the kernel to optimize specifically for embedded systems. It is straightforward to compile a kernel that consists of only the neccessary drivers and features, resulting in the kernel being fairly small in binary size and using few resources. Opting to use Linux in an embedded system does increase complexity as opposed to using a plain microcontroller. However, this increased complexity must be justified to use an operating system, such as Linux. Some reasons for choosing Linux are its solid network stack, robust file system, various I/O interfaces parallel processes and threads.

Although understanding of operating system internals is not required, it is helpful to understand the functions and internals of operating systems. Further understanding is a requirement in order to develop device drivers. In Linux, there is a two-level separation of priviledges: kernel space (or kernel mode) and user space. Kernel code and all kernel drivers are executed in kernel space, where they have unrestricted access to all physical devices. For security reasons all normal software, that is, actual processes being executed by a user, are run in user space. User space is an isolated and safe environment for executing user applications, where an application can utilize the higher level services provided by the kernel. These services are for example editing files and connecting to a remote resource over Transmission Control Protocol/Internet Protocol (TCP/IP). The communication between user space and kernelspace is done with so-called system calls, or syscalls. System calls, in Linux, are executed inside interruptions, meaning when user application wishes to call kernel service, it raises a hardware interruption and execution is handed over to kernel interrupt handler. The concrete interrupt calling is usually abstracted away from a user application through standard libraries, for example with LibC.

Once Serial Peripheral Bus (SPI) bus is enabled in kernel configuration and boot time parameter, they become available to user space through an SpiDev driver. By default,

the SPI devices are communicated with a character device interface. The character device interface abstracts the I/O communication to simple file operations using syscalls `open()`, `read()`, `write()` and `close()`. A Character device abstracted SPI devices are located in `/dev/spiX.Y`, where X is the SPI bus number and Y the device number. Further configuration for an SPI bus is possible using `io_ctl()` syscall. [15]

This is a straightforward and easy way to transfer bytes to SPI devices. Chip select is handled by kernel driver and is chip select pin normally dependendent on Central Processing Unit (CPU) model. Using character device interface, however, requires an additional syscall to be made in order to communicate with the device. This causes an additional context switch making subsequent SPI transfer delays undeterministic. Hence, for high-speed or periodic SPI transfers it is best to create a kernel driver, which can communicate with the device asynchronically from user application.

In a properly configured Linux kernel, both General Purpose Input Output (GPIO) and SPI are visible to user space, meaning that an application developer does not need to write a kernel driver for devices that utilize SPI or GPIO. This is a great benefit, since the application developer can now rely on the services provided by the kernel, namely security and isolation. But it has the drawback that in the user space, there are no deterministic delays and timings between system calls and I/O transfers. Thus, for accurate timings, either kernel driver, Real-Time kernel, or microcontroller is needed for e.g. deterministic delays and sampling [16]. This is a clear constraint when it comes to implementing certain features, such as signal processing and Finite Impulse Response (FIR) filters that operate on data coming from GPIO or SPI bus.

2.4 Python 3 and Qt Framework

Python is a high-level interpreted programming language that has been initially released in 1991 and has been in active development ever since its initial release [17]. As of writing this, the latest stable Python release is version 3.9.5 [18]. Python supports multiple programming paradigms, ranging from functional to Object-Oriented Programming (OOP) [19]. The Python's runtime environment uses garbage collection as its memory management strategy, meaning that the interpreter runtime is responsible for reserving and freeing the memory as the program is executed [20]. Depending on the use, it may be beneficial to let the interpreter do the memory management, trusting that it does its job better than the developer would be able to. There are certain situations, however, where the garbage collector can be problematic, or less optimal. An example is an image data, which is essentially a large block of memory or a buffer. Using pure python to store raw data would result in losing the benefit of locality, since Python does not necessarily store data in blocks, but scatters the data accross the available memory.

Python has a great support for calling C-functions directly from Python, and a geat part of all the libraries available does use C-functions to perform CPU or Memory intensive operations. In such cases, the Python runtime does not manage the memory regions

the C library allocates itself, allowing certain optimizations, such as hardcoding buffers to operate on and pointer arithmetics. Many existing libraries and framework, such as Qt, Gtk have one or multiple libraries for using them directly from a Python script.

Qt is a Graphical User Interface (GUI) and application framework built with C++ [21]. It has bindings to many languages, including Python. Qt uses C++ containers and its own smart pointers for managing memory use, where a parent class always takes responsibility of all of its child objects [22]. This is in contrast with how Python manages its memory using the garbage collector. Unreferencing a Qt's QObject in a Python script will not free it from memory, but makes it inaccesible from the Python script [23]. This may easily result in memory leaks when using such an environment, which has two methods for managing the program's memory. In such case, careful design and implementation is in order when sharing data between functions or threads.

Unlike some other, more modern programming languages, Python is not originally designed for multi-threaded programming. Modern versions support multi-threaded programming and have basic data synchronization. Python 3 introduced asynchronic code execution, which essentially contains its own scheduler and runs multiple tasks in a single Operating System (OS) thread. It is especially well suited for I/O-bound systems, where significant delays would otherwise starve the execution. [24]

2.5 Image Processing

Signal processing, or more commonly known as Digital Signal Processing (DSP) , is the operation of modifying and filtering digital signals [25]. Digital signals are always discrete numeric values, where sampled data then represents the original signal. DSP is widely used in digital systems since numerical signals are very rarely operable and require further reduction of noise and other defects.

Image processing is the term used for various techniques for processing digital images. Like many signals, Electrical signals are often 1-dimensional, where for instance ADC voltage is measured and sampled over a range of time. Image processing differs from most other DSP techniques in that an image is always 2-dimensional (grayscale) or 3-dimensional (color) data. For humans it is very easy to interpret and read a graphical image, but for a computer to identify abstract shapes such as rectangles or circles, not to mention human faces, is nothing short of difficult.

Using various techniques the original 2d image data must be processed to find abstract features, such as geometrical shapes. Today, one of the most prominent most challenging uses for digital image processing is face detection and recognition. Most of the methods involve the use of machine learning techniques, ranging from simple algorithms such as Linear-Discriminant-Analysis (LDA) and Gradients to complex and deep neural networks [26], [27], [28]. Today, for more advanced use cases, such as face recognition, a neural network based processing might work better and has clear benefits, when enough training data exists [28].

Manipulating and processing 2 or 3 dimensional image data in practice requires matrix operations. The operations themselves are not complex. For instance, blurring an image with gaussian operation requires a convolution operation using a suitable binary matrix, or a kernel to operate the image with [29]. Calculating matrix operations is usually a sequential operation, which means the operation may take a relative long period, especially when done with slower CPU. This is something that may need to be taken into an account when developing image processing algorithms, especially if run with low-end CPUs, or if the program is otherwise unable to execute parallel threads.

For successfully extracting important features, a customized pipeline, or a chain of filters, must be crafted. Because of the nature of the problem, these pipelines are nearly always customized and mandatory for getting acceptable results. The processing pipeline takes an image as its input, which has a fixed size and is either grayscale or color image. The output can either be another image produced from input, or a feature vector containing the description of the objects in the input image.

3 SPECIFICATION

Before the calibration system's software can be designed and built, the requirements and specification must be defined. This is to both enable effectively constructing the software, and to work as an acceptance criteria for the completed software. Writing the specification is one of the most difficult parts for any project, since the balance between level of details and general lines greatly affect the outcome and may limit the options for the implementation. With a good specification, the implemention does not suffer from unneccesary constraints. In addition, a good and unambiguous specification is easy to verify.

3.1 Medical system

The medical system's requirements originate from three parties: regulation as well as both an external and an internal customer. Regulatory requirements define the overall framework where the product must operate in. In addition to the regulatory requirements, an external customer sets additional requirements, such as suitability for a medical use, as well as a simple interface to operate the device. The internal customer has requirements for the performance of the calibration software in terms of speed, quality, repeatability and traceability. Throughout the medical laser system's product development phase a so-called stage-gate process was being used [30]. During the development stages the laser system's system specification was defined. The laser system's specification dictates the requirements for the calibration system as well.

The medical system's architecture is shown in Figure 3.1. As such, the system consists of three physical components: a laser device unit, an optical beam shaper unit and an external user interface. The user interface is not visible in the Figure. The optical beam shaper unit is connected to the laser device unit with both an optical fiber and electrical wirings for communicating with the optical beam shaper unit. The fiber is secured to the optical beam shaper unit and cannot be disconnected without losing calibration accuracy. Thus the fiber must be considered as part of the calibration data for the optical beam shaper unit. For operation, the fiber is connected and fastened to the laser device unit. For interoperability between different and laser drivers, each of these components must be characterized separately. By carefully selecting the set of calibration parameters, it is possible to fully describe the function of both the laser device unit and the optical beam shaper unit in relation to each other.

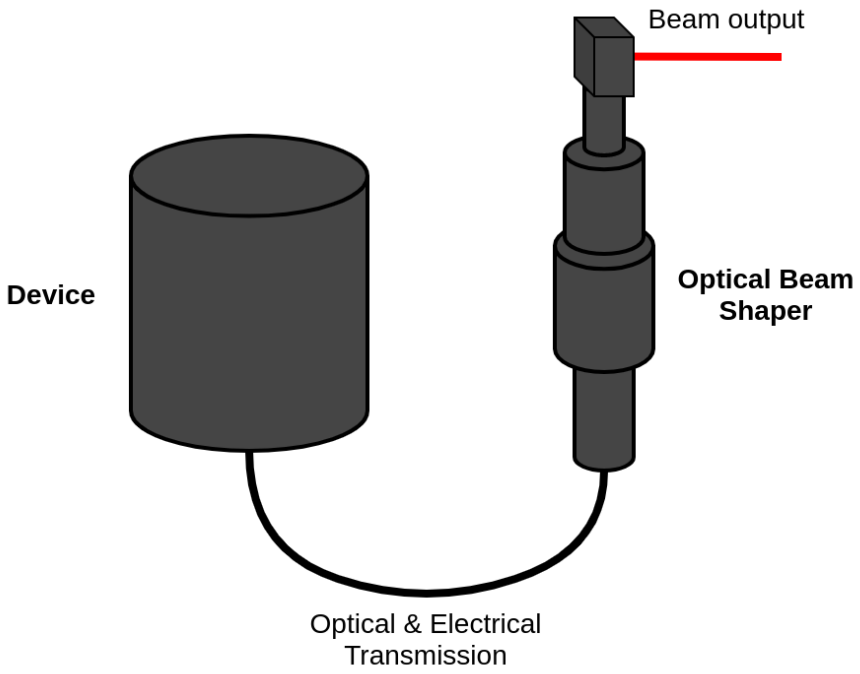

Figure 3.1. *Medical system architecture*

The laser device contains two laser modules, an aiming beam and a treatment beam. Aiming beam is a low-power laser, whereas the treatment beam is much higher power laser. Also, the two laser modules produce different wavelengths.

3.2 Requirements for calibration

The automatic calibration system's requirements originates from all of these upper level requirements. In order for the medical device to operate as specified, approximately 10 different sweeps and a few per-device constant voltages and resistances need to be measured. This results in a dozen of two-dimensional tables, which the device needs during operation. These characterization tables are then stored both locally to the device's and externally to the company's internal storage. Various electrical characteristics need to be measured from both the laser device and the optical beam shaper unit. In addition, some characteristics, such as an LIV curve, need to be measured for both a treatment beam and an aiming beam, taking the optical beam shaper 's effect into an account. Since the treatment beam and aiming beam use different wavelengths, they have different responses in the optical beam shaper unit.

The overall system requirements are presented in Table 3.1. To summarize the system specification, the requirements state that the treatment beam and aiming beam powers, the spot size and their respective photodiode-readings have to stay within the required limits throughout the operating range. For implementing the requirements, an automatic calibration system was developed. This calibration system contains various in-house built electrical equipment for characterizing the laser device and the optical beam shaper

separately from each other. In addition, multiple commercial meters and mechanical and optical components are used. The physical calibration system is outside the scope of this work, which focuses on the software implementation of the calibration system. During the lifetime of the medical product, some characteristics might need further tuning and, thus, a periodic mainentenace is in order. The calibration process is then a part of the manufacturing process of the medical device. In addition, full calibration is done during a periodic maintenance.

Table 3.1. *Product Requirements*

Property	**Requirement**
Treatment beam irradiance	Within ±15 % of the target power
Aiming beam irradiance	Within ±35 % of specified output power
Spot size readings	Both absolute and relative errors below specified limit
Spot standard deviation	Below 15%
Spot output sharpness	Below 5% of power inside 90-10% transition area
PD power reading	Below ±50 % of actual output power
Interoperability	laser device and optical beam shaper must be interoperable

3.3 Requirements for calibration system

In addition to the requirements for the medical device and its specification, the calibration system has some additional requirements, both for ensuring the validity of the calibration output and for company's internal purposes. Further requirements for the calibration software are presented in table 3.2.

Table 3.2. *Calibrator system's requirements*

	Requirement
1.	System can measure laser device internal reference voltages and pullups
2.	System can measure optical beam shaper resistances
3.	System can characterize optical beam shaper unit's losses
4.	System can emulate optical beam shaper electrically to laser device
5.	System must have persistence for both raw and post-processed data as well as for device configurations
6.	System must be capable for possible future remote operation feature
7.	System must be composed of modular and reusable components
8.	System must have a graphical user interface
9.	System must be safe to use

Each calibration step is specified separately along with any additional requirements, such as saving the characterization data to the medical device or to external

persistence. Certain characterizations set additional constraints for the contents of numerical characterization data, such as columns being monotonic or having upper or lower column-wise limits.

4 IMPLEMENTATION

This chapter describes the overall architecture of the system and some details and components of it. The intent is to implement a system that fulfills the specification presented in previous chapter. The development process for many calibration processes required several iterations, due to the fact that the calibration process itself was being developed at the same time, and from time to time changes were needed to existing logic. Thus close feedback loop helped to keep the project on track.

4.1 Software architecture

The design of the calibration software started by outlining the software architecture. Throughout the implementation process the architecture was refined and certain interfaces between the components were fixed. The overall architecture of the calibration system software is shown in Figure 4.1. Although the calibrator system is specific to the medical device it calibrates, many of the components developed can be used in various similar projects, ranging from device calibrations to product development. Thus a modular design with clear separation is beneficial.

A central idea was to use state machines for each calibration sequence. As the language that the software is written with is Python and Qt framework was being used, it made sense to build the software using OOP -paradigm. By encapsulating all logic in classes, it is possible to create a modular software, where ideally changing functionality requires rewriting only that specific component, or inheriting a new subclass for new functionality. The good thing about OOP is that by definition the logic is built as a tree consisting of inheriting classes.

In addition to the calibration state machines, a complete calibration software needs many additional components to work properly. This includes application configuration, a GUI, pre-compiled driver libraries from from 3rd. parties, and other application bootstrapping techniques. In this project, the configuration data is stored in an `ini`-formatted text file. Using a text file makes parsing the file easy and allows the end-user to easily modify certain parameters, if needed. Such parameters include not only values for processing and limits, but also calibration constants and calibration due dates. In addition to easy editing, a text file is easy to store in a version control system, such as a git repository.

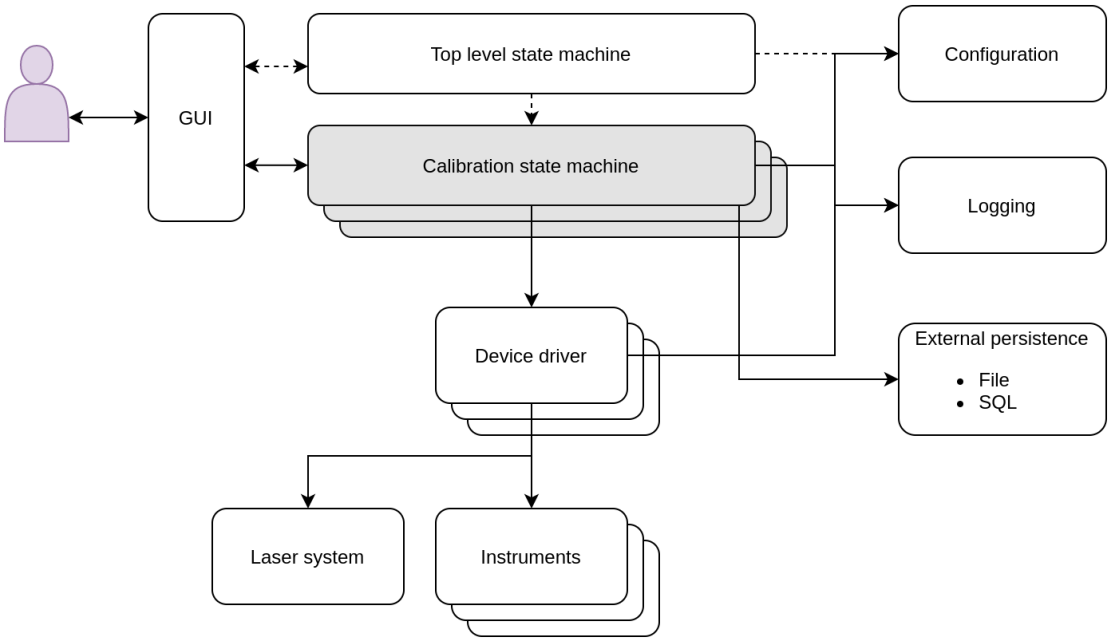

Figure 4.1. *Software architecture*

In addition to using OOP, it helped to separate all functions by their level of concern. A central concept was to use Model-View-Controller -paradigm (MVC) for separating the logic from user-interaction. Throughout the development process multiple refactorings were necessary to further abstract the logic. For instance, an abstract calibration classes were made in an object-oriented manner and most of the drivers share interfaces and abstractions for using IO-threads for communication. Although the Qt framework does recommend a variation of MVC, using MVC it is possible. When using Qt with Python, because of the dynamic typing Python uses, using Qt objects and types are actually loosely coupled in the Python runtime environment. This is both a great benefit, as it imposes great freedom, but also a potential risk for runtime errors. The controllers communicate with the devices through drivers, and controllers contain all the business logic. View layer, consisting of widget-based components and complete views, is used for graphical user interface and all user-interaction. This has the clear benefit that if the interface is to be changed to another technology, for instance to a web-based interface, it would require rewriting only the view layer. Or if the program was decided to run completely on command line and not have graphical user interface at all, that would be possible as well. The internal signaling uses QT signal and slot -objects, which work well in scenarios like this where there are multiple producers and consumers for the data. In addition, using signal / slot mechanism to dynamically connect the components to each other allows running unit tests for each component.

4.2 State machines

Most calibration procedures follow a simple state machine architecture, as depicted in Figure 4.2. The state machines in this project are implemented using Python 3's asyncio-library and th Qt framework. The complete calibration software contains multiple state machines, one for each calibration sequence. In addition, drivers often contain their own state machines, GUI may include a state machine, and a top-level application has one too.

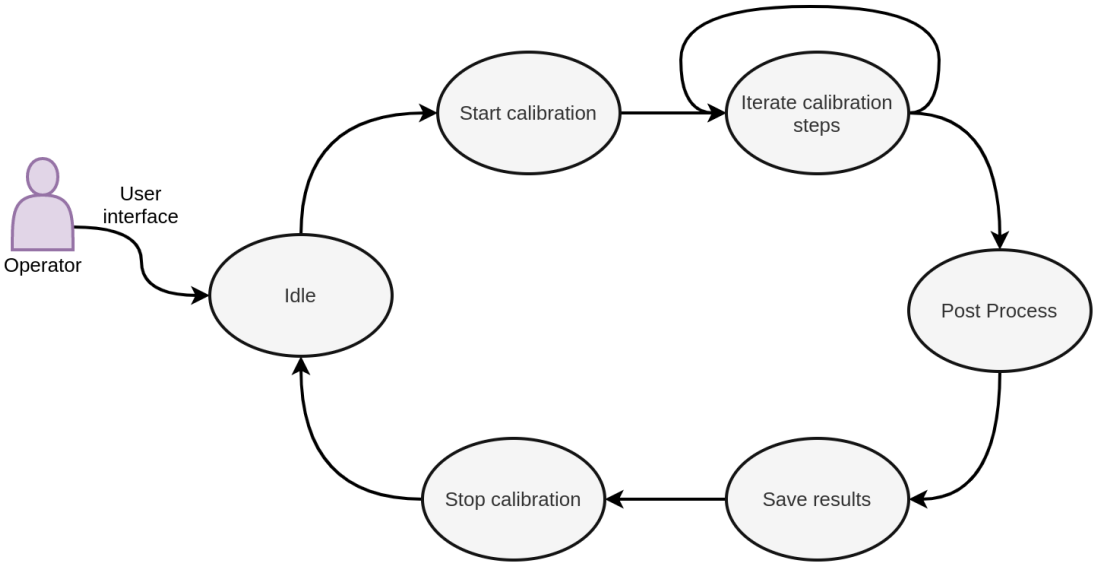

Figure 4.2. *Calibration state machine*

The first state in each calibration state machine is the idle-state, since there is no calibration in progress yet. The next phase is to start or initiate a calibration. This may require manual intervetion from user, such as connecting correct equipment to the medical system that is being calibrated, setting correct optical filters if needed and creating initial values that are then further optimized throughout the calibration. Also an important aspect in starting each calibration is asserting the correct configuration in order to ensure a safe and effective calibration. Such assertions may include reading existing configuration and verifying that the meters and optics work as expected. Then, an iterative step loop is exectued to either sweep over power or other range, or iteratively optimize certain factor. Sometimes the parameters of interest cannot be directly measured, but must be derived from other measurements. In such cases, some sort of sweep is performed and a linear or nonlinear regression is applied for calculating the parameters. It is possible that iterative steps fail, either due to a user error, a faulty device or other error. In case of error, the calibration must be stopped in a controlled manner, ensuring that the laser is always turned off and other equipment are released as well. After successfully iterating the calibration steps, the state machine returns to idle state and further post-processing can be applied. After post-processing the raw data, the results are saved and persisted both to the device's internal memory as well as

to the company's internal storage.

As for the device drivers, they usually contain a simple state machine for initialization, communication and deinitialization. However, due to the nature of I/O communications, the state machine may need to cover asynchronic calls between the physical device. For example, when using serial transmission, where a message may arrive asynchronically after calling, a special logic must be included in order to match the transmitted and received messages after parsing the messages and identifying separate message packets.

The application level state machine is similar to a driver state machine, in that it only has states for initialization, normal operation and deinitialization. During the initialization phase, all hardware drivers are initialized and configured prior to proceeding to normal operation. The same applies to deinitialization. In addition, normal operation state consists of states idle and calibration. This acts as a synchronization mechanism to ensure that only one calibration can be run at a time. Since many calibrations use multitude of peripherals, it is insufficient to only synchronize access to each device, so the synchronization must be handled in a higher level of abstraction.

4.3 Asynchronic device drivers

One of the important questions when developing an application with I/O-operations, let alone communicate with dozen of devices, is to decide how the application manages the execution flow of the program. A traditional solution for I/O-operations is to use a callback-based executio flow. The trouble with callbacks is, however, that handling the state as well as synchronizing acccess to the data gets complicated and error-prone very quickly. Thus the architectural desicion was made to use the Python's `Asyncio` library. `Asyncio`'s eventloop executes in a single thread, providing a concurrent execution for asynchronic tasks. `Asyncio` does not provide real CPU-level parallel execution, because all of the tasks are scheduled to be run in a single OS thread.

Inside the `Asyncio` eventloop, any device or timer that requires action is then handled concurrently. User interactions are also handled inside this eventloop and actions are dispatched for device operations as needed. While `Asyncio` generally works very well in I/O-bound systems, it does incorporate additional challenges. First, connecting the Qt runtime to Asyncio had to be solved.

All drivers that communicate with an I/O-bound resource, including remote database connection and storage, are synchronous by design. Synchronous, in this case, means that the function call will block the program execution for as long as the actual communication with the device takes place. Asynchronous function call, in contrast, hands the execution back to the eventloop executor and lets the eventloop execute other tasks, while actual communication with an I/O-bound device is waiting for completion. I/O-bound devices always create delay, ranging from milliseconds to seconds. For instance, a single Serial Peripheral Interface (SPI) transfer takes a couple of

milliseconds, while database query may easily take one or more seconds to complete. In such scenarios, the main thread and user-interface thread cannot afford to wait that long, and drivers must communicate with other resources during those intervals. First, because of external constraints and timings, physical devices need may need continuous exchange of data for preventing buffer overflow or other execution errors. Second, the end-usre may not tolerate the application being non-responsive for most of the time. Hence, asynchronic drivers must be implemented to allow the overall system to operate properly. If `Asyncio` library was not available, a callback-based system would have been needed. This would have complicated the overall architecture and implementation of the program significantly.

In this project, with Python 3 and `Asyncio` library being available, each function call must conform to `Asyncio`'s concurrent calling convention. In practice, each all blocking call must be encapsulated with await-keyword or use `Asyncio` futures. An overall architecture for asynchronic calls using synchronic I/O-calls is presented in Figure 4.3.

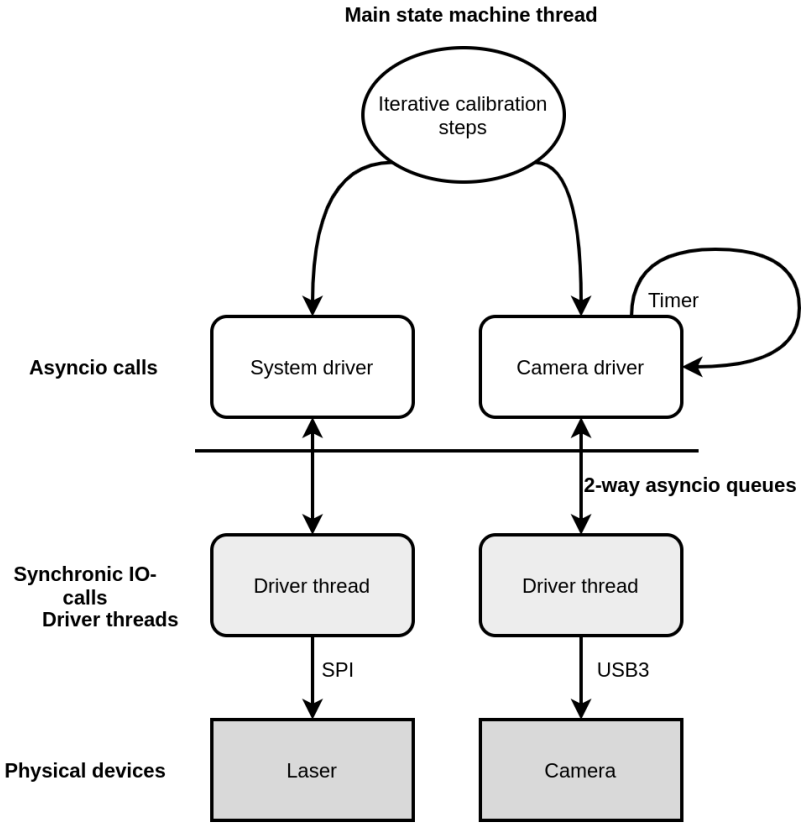

Figure 4.3. *Software architecture for asynchronic & synchronic calls*

By using this technique, it is possible to manage the calibration as a single state machine that communicates with multiple devices in an asynchronous manner, while still using one thread for the application execution. Drivers naturally contain separate threads for low-level communications. This ensures that necessary timeouts are never exceeded and physical devices function as intended.

4.4 Calibration system safety

Since the calibration software controls physical equipment, including using a relatively high-power laser, it is mandatory that the software is safe to use. Safety, in this case, means that the software will not send illegal or bad control commands to any equipment, use the laser on without direct control over it or send any other command that might damage the laser system or any equipment. The greatest risk related to the calibration process and the calibration system is the laser device itself and its optical output, which can easily lead to eye damage if not properly protected against. While the operator must use safety goggles during operation to prevent eye damage, careless operation or faulty functioning may still lead to e.g. device or system damage. Thus it is imperative that the calibrator system always shuts down the laser, regardless of any errors during the operation or calibration procedures.

Additional steps were taken for ensuring these requirements. For instance, the contains timeouts for turning the laser off. Also a graceful shutdown is implemented in case any runtime errors occur that might crash the application. The best way to ensure such operation is by abstracting the calibration procedure to a separate class, which enforces a stop function that is called after any calibration, be it failed or successful. `Python` includes try-except syntax and a `contextmanager` for ensuring that shutdown sequence is always executed. Additionally, a clear indicator must be shown to the operator of any laser activity.

4.5 Software Licenses

As with almost any software these days, the implemented calibration software uses numerous 3rd party libraries, most of which are licensed under open-source licenses. These licenses grant a user the right use, modify and share these libraries. Licenses might have additional constraints for usage as well and distributing the software. The most common open-source licenses are GPL and its variants, Apache and MIT license. The most important libraries and their licenses are collected to Table 4.1.

Table 4.1. *Open source licenses for libraries used in calibrator software*

Library	**License**
QT 5	GPL3
PyQt5	LGPL3
Scikit-Image	Custom
OpenCv	Apache 2
Numpy	BSD-3
Scipy	BSD-3

Pyqt5 and its license GPLv3 being the strictest, renders the whole project as a GPLv3 licensed project. This imposes the additional requirement that if the calibration system and software is ever to be distributed to external parties, according the GPLv3 license the full source code must be available along with distributed software. Since the calibration system is not intended to be published or shipped to any 3rd parties, the licensing does not add any additional risks.

4.6 Device drivers

An important level of abstraction in the working calibration software is the device drivers. Since the the calibrator system aims to measure physical quantities and control physical devices, the drivers are a mandatory layer for communicating with devices. Depending on the physical device, a few communicating mechanisms are used, such as serial communication, SPI and a General-Purpose-Input-Output (GPIO) are used.

4.6.1 GPIO

There are primarily two ways for using GPIO directly from userspace. First, the GPIO pins can be memory-mapped with `mmap`, after which they are available through normal file system operations. Memory-mapping and sysfs operation is shown in Program 4.1. The exact GPIO identifier is always platform-specific. This is a very convenient way to prototype and develop peripherals and external devices, but may not be suitable for production use.

```
1  # reserve GPIO pin 10 for use:
2  echo 10 > /sys/class/gpio/export
3
4  # set pin 10 direction to out, making it available for & writing:
5  echo out > /sys/class/gpio/10
6
7  # read input pin, file contains '0' or '1':
8  cat /sys/class/gpio/10
9  # write output pin, either 1 or 0:
10 echo 1 > /sys/class/gpio/10
11
12 # when done, free the pin:
13 echo 10 > /sys/class/unexport
```

Program 4.1. *Using GPIO in Linux through gpio sysfs*

A second way to use GPIO from userspace is with the `io_ctl()` syscall. While the syscall can be invoked directly from Python, multiple libraries exist to abstract it for direct use in Python. In this project, a library called RPi.GPIO [32] was used to directly read and write GPIOs from Python driver.

```
# factory is a singleton object, only one instance exists
board = gpio_factory().init_board(board_name)

# initialize pins
gpio_input = board.get_input(input_pin)
gpio_output = board.get_output(output_pin)

# gpio is now ready to use:
input_value = gpio_input.read()
gpio_output.write(1)
```

Program 4.2. *GPIO board and pins using a factory pattern*

The RPi.GPIO library routines were abstracted behind a factory-pattern. The intent was to ensure a modular design and to make porting the application to another platform an easier task to accomplish. The driver factory then, combined with an abstract gpio_board, gpio_input and gpio_output classes, has an interface and workflow, as shown in Program 4.2.

4.6.2 SPI

SPI was used between ADC, DAC circuitry and the PC host system. RaspberryPi provides two or more SPI buses, depending on the model being used. Since the GPIO pins have alternative functions as well, the pins must be explicitly reserved for SPI. This is done by configuring the pins in the Linux device tree data structure. While this can be done manually, it is easier to do with RaspberryPi's hardware configuration file, `/boot/config.txt` [33]. On RaspberryPi, the `/boot/config.txt` acts as an overlay description for manipulating the underlying Device Tree. During boot, the overlay is combined and as a result, a complete device tree description is passed to the Linux kernel, containing the edits included in the `/boot/config.txt`.

While implementing the SPI communcation with ADC and DAC circuits, there were issues related to SPI bus clock phase and polarity. The problem was that the DAC and ADC circuit used different settings, but they were connected to the same SPI bus. The solution was to either do a hardware fix for inverting the bus polarities, or fix it using a software function wrapper, both of which were tested in the propotyping phase. Eventually the polarity issue was resolved using software wrapping function for setting correct SPI bus settings while communication to each converter unit.

A Python package `spidev` was used for communication. The package wraps `SpiDev` Linux driver and uses it as an underlying library. The usage is straightforward. After opening the bus and the given device by its address, data packets can be sent and read. It is even possible to modify the bus settings between packets, which was required for communicating with different peripherals.

4.6.3 Camera

Industrial cameras are widely available with two interfaces: USB3 and GigE. Univeral Serial Bus version 3 (USB3) is a conventional USB-connector with a guarantee of enough bandwidth for real-time video. Gigabit-Ethernet (GigE) is another solution, that is useful especially in scenarios where there are multiple high-speed devices, such as cameras, and possibly even long distances and wirings between the devices. Due to its simplicity, a camera with a USB3 interfaces was selected. The only problem with USB3 version of the camera seems to be an unreliable initializion and deinitializion when plugging it into the PC. This can be overcome by reattaching camera and trying again, which in this case is acceptable. While this is a minor problem, it is inconvenient for the end-user to reconnect the camera for resolving the issue. For a more autonomous scenario, though, this kind of unreliable behaviour would be unacceptable.

The camera manufacturer provided a low-level compiled driver library for operating the camera, which contains both C and Python interfaces for communicating with the camera. This driver implements communication with both USB3 and GigE cameras, meaning that changing the camera model is possible later on. The usage is relatively simple: initialize the camera, configure the exposure time if needed, grab an image or configure a continuous grab mode, enabling stream of images. After getting data from the camera, an image manipulation pipeline is utilized for computing a feature vector containing the description of the laser beam.

4.6.4 Spectrometer

The spectrometer uses a USB3 interface for communication and a low-level driver is supplied by the manufacturer. In addition to `init()` and `deinit()` the driver contains functions `take_measurement(time_ms)` and `take_dark_measurement(time_ms)` and a few more methods. Because a spectrometer has inherently background noise in the measurements, it is useful to reduce the background noise by taking an average of multiple samples that do not contain any actual signal. This so-called dark measurement is then subtracted from each sample. Depending on the spectrometer the dark measurement reduction is either automatic or manual operation.

The spectrometer operation did not include any difficulties besides handling the data. The data is a 1-dimensional array of illumination in each point of the sensor. Depending on the spectrometer, a span of 1 nm may contain dozens or hunders of measurement points. Also an automatic exposure control was implemented for assisting the end-user. After these functionalities it is possible to measure the spectral characteristic of the laser beam. Interesting factors are the distribution of the energy over wavelength as well as the peak wavelength.

4.7 Host system

As the calibration system contains not only software, but in-house built electronics and equipment as well, the host system plays an important role too. Because the low-level interfaces, such as SPI and GPIO, were needed in this project, it was concluded that Linux was the optimal operating system. A PC can be any computer that supports these peripherals. To make the prototyping and development easier, a RaspberryPi was chosen as the computer unit. Should there be a good reason, RaspberryPi can also be easily replaced, given the new unit has the same peripherhals available. For performance reasons, an external SSD hard drive is used for root filesystem. The Boot partition must reside in an SD-card, though.

Another important decision to make is selecting a Linux distribution to use. Since all distributions use more or less the same Linux kernel, the distribution mostly affects the environment and tooling, such as package managers and kernel version. Usually a custom distribution is built for embedded environments. However, building a custom distribution did not provide any benefits for this project, but whould have required a substantial amount of additional work. For this reason, the RaspberryPi's official distribution, Raspbian, was selected. Raspbian is an easy to use, widely available and a well supported distribution, It is based on the latest stable Debian, so there's a good level of security for stability. For configuring the hardware peripherals on RaspberryPi, the GPIO and SPI peripherals needs to be enabled. While this can be achieved with command-line tool `raspi-config`, peripherals are managed the easiest through RaspiberryPi's boot overlay filesystem, available in `/boot/config.txt`.

In order to be able to actually use the calibrator software, the host system needs a network connection. This includes configuring the DHCP settings and a possible VPN access. Additionally, a display is needed. Because the Qt window is used as a GUI, the end-user must be able to view the X window, either over VNC connection, X-forwarding the window with SSH or by physically attaching a display. The final thing to configure and install are the device drivers for e.g. SQL database and USB devices, such as camera and spectrometer. Due to the nature of how permissions are managed in Linux, all devices are going to need separate handling for allowing user to access them. This is done easiest by creating a separate unix user account and assigning devices to that account using suitable udev-rules, residing in `/etc/udev/rules.d/` directory. Vendors may even provide their own udev-rules to aid the setup of their products.

4.8 Profiling The Laser Beam

Beam profiling is the operation of measuring the laser beam shape optically, typically with a camera. Since a laser ideally has only one discrete wavelength it emits at, that can measured with a spectrometer, the color data of the beam profile is irrelevant. Using a color camera, which uses an optical filter called a Bayer array for capturing color data,

would in fact, decrease the quality of the raw image, because a Bayer filter reduces the amount of light that the sensors receives [34]. Hence, a grayscale camera with a large enough sensor area was selected.

The laser output is pointed directly to the camera sensor, through some attenuation with ND-filters. For optimal results and minimal distortion, it is best to not use any optics, if possible. The optical beam shaper unit contains optics for the laser beam, so it was concluded that the camera does not need any additional optics.

For this project, the product specification in Table 3.1 lays out the requirements for optical inspection for the laser beam. In this case, the laser beam can be considered to be geometrically circular object in the center of the image. One parameter to measure is the actual circularity, which might not only be elliptical but have defects and a certain roundness as well. Thus measuring ellipticity, though sufficient, is not a perfect measure for the circularity. The simplified pipeline that was developed is depicted in Figure 4.4. The feature vector contains the beam diameter, the relative circularity, the standard deviation of the beam intensity, the mean brightness and a few debug values such as threshold values.

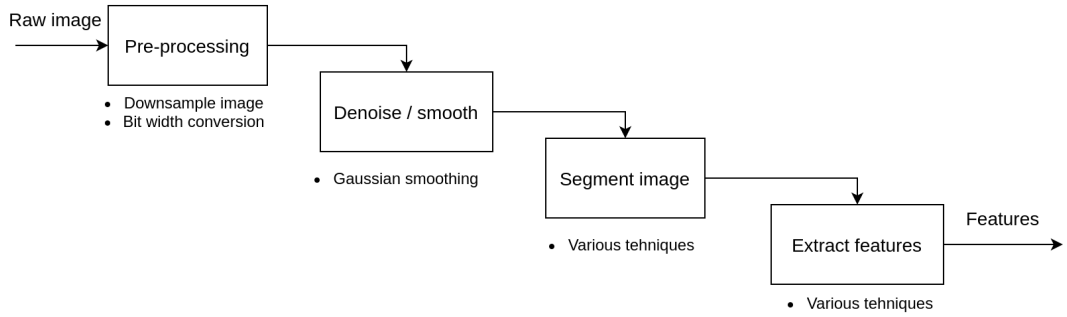

Figure 4.4. *Simplified beam profiling pipeline*

The beam profiler, which is written with Python 3, uses numerical and machine vision libraries such as SciKit-Imaging as well as OpenCV. Both of these are widely used libraries for image manipulation. From these two, OpenCV is probably more performant, especially in low resource environments. SciKit-Image, on the other hand, is better suited for prototyping, testing and developing image processing algorithms. It is expected that the whole pipeline must be refactored to only use OpenCV in the future. The processing pipeline's development process is largerly trial-and-error. Systematic approach helps, and the problem must be split into smaller pieces.

4.8.1 Processing Pipeline

The image processing pipeline consist of multiple distincts stages, where each one uses its own algorithm or multiple algorithms for achieving the results. The first step in processing the image is downsampling it. The necessity for downsampling depends on environment and computer resources. With RaspberryPi, it is necessary to downsample the image to 1/4th or 1/8th, at least for the segmentation of the image. Following downsampling is thresholding the image and segmenting it based on the previously determined threshold level. The purpose of thresholding is to find an initial value to separate the beam from background in the image. The initial threshold does not need necessarily need to be correct, but a good initial value decreases the time an iterative approcash requires for finding best threshold value. The better the initial guess is, though, the better the segmentation will be, as well. As for thresholding, SciKit-Image package `skimage.filters` provides various algorithms, such as Li, Otsu, and local thresholding [35]. Most of these algorithms are covered in [36].

After the threshold value has been determined, the next step is to segment the image into uniform areas and to find edges between the regions. Segmentation simply means that the image is split into multiple regions, where each region represents an object or a partial object. In this project, the image is segmented into three regions: the background, the beam edge region, and the uniform beam region. The distinction between the beam edge and the beam region originates from the optical beam shaper unit's specification, where the laser beam is said to have both the beam area itself and a lower-power outer region. The segmentation can happen with any number of filters combined. Software libraries often contain functions for finding contours or edges in various cases. Many of these existing solutions were tested, such as finding circles using Hough Transformation [36], [29], but it proved to be insufficient where there are either aritifical reflections in the image, or the laser beam is not completely circular.

A sample view of the thresholding and segmentation is presented in Figure 4.5. In the figure, markers are drawn in the eventual threshold. Anything over 85% is interpreted as the beam top area, and the region between powers of 5%-85% is interpreted as the beam edge area. Edge area is a circular belt around the beam top area. Various features are then calculated around these two regions as well as between their relative size and power they contain.

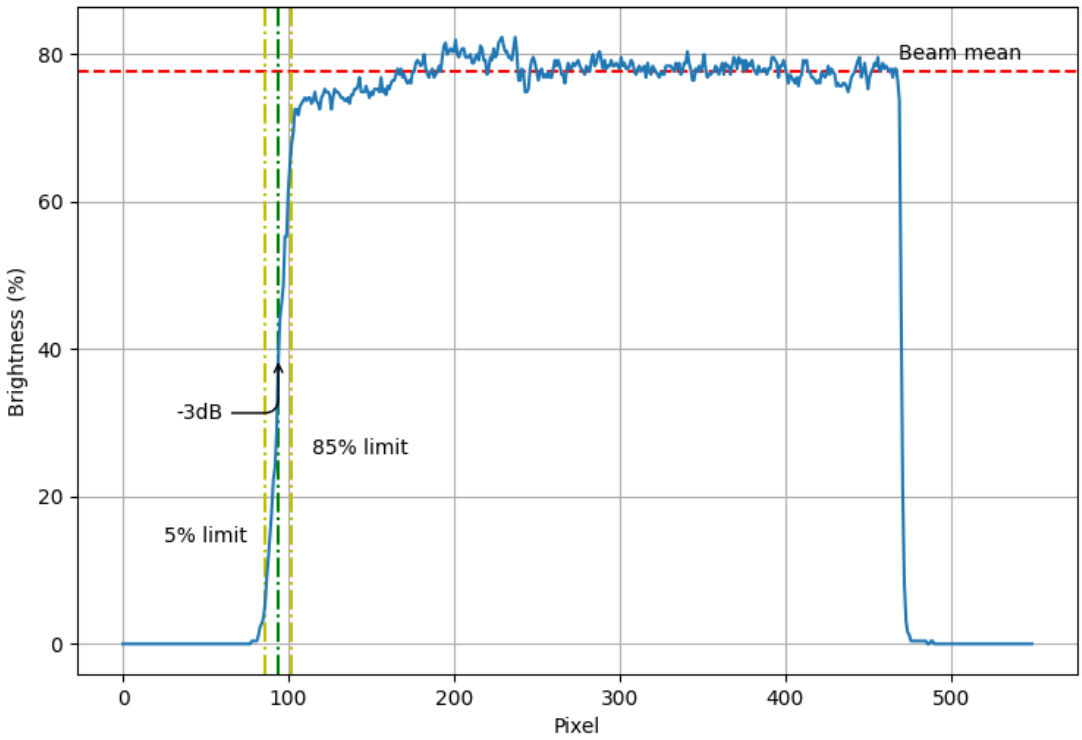

Figure 4.5. *Beam profiler cross-section of the beam*

Sample images of the beam profiling can be seen in Figures Figure 4.6 and Figure 4.7. Both figures show the original grayscale image in left, which is colorized to mark the brightness of each pixel. On the right is same beam but with the segmented beam edge mask drawn on with white color. In addition, the calculated center of the beam is marked with white circle. The beam edge area is drawn with white, while the beam area is drawn with gray color. Figure 4.6 depicts a larger laser beam, where the beam is nearly perfect circle. Figure 4.7 on other hand depicts a smaller laser beam, where the shape is not as close to a perfect circle.

A laser beam has one particular trait, known as the speckle, that might seriously affect the results of the beam profiler and must taken into account. Speckle is the high-contrast, grainy, binary-like, noisy effect inside the beam, that naturally occurs in lasers due to the interference of the coherent light with itself [37]. Since one of the purposes of the optical beam shaper unit is to minimize and level out the speckle in the laser beam, most of the noise has already been removed from the laser beam. Thus, further noise reduction or high-contrast elimination is not necessary. Some level of speckle is still present in sample Figures 4.6, 4.7 and 4.5.

The beam profiler also includes necessary automation, such as an automatic exposure control. The intensity inside the laser beam in the image is much higher than the background noise outside the beam, i.e. there is a high contrast and a clear separation between the background and the laser beam in an image. Priority is given to the beam area, which should be well exposed but not saturated. This may be difficult to achieve,

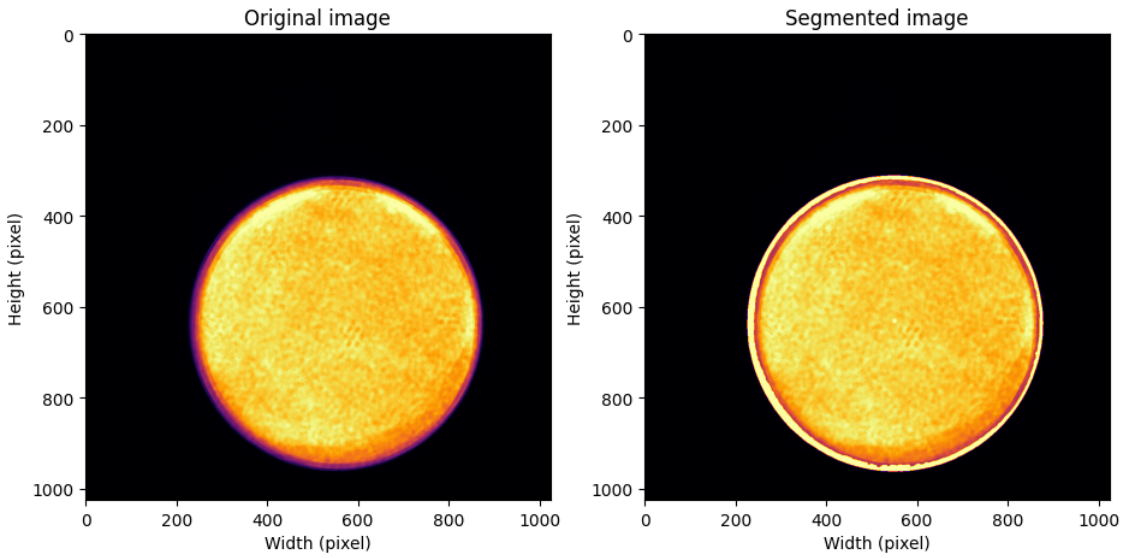

Figure 4.6. *Sample of big spot size and its segmented mask*

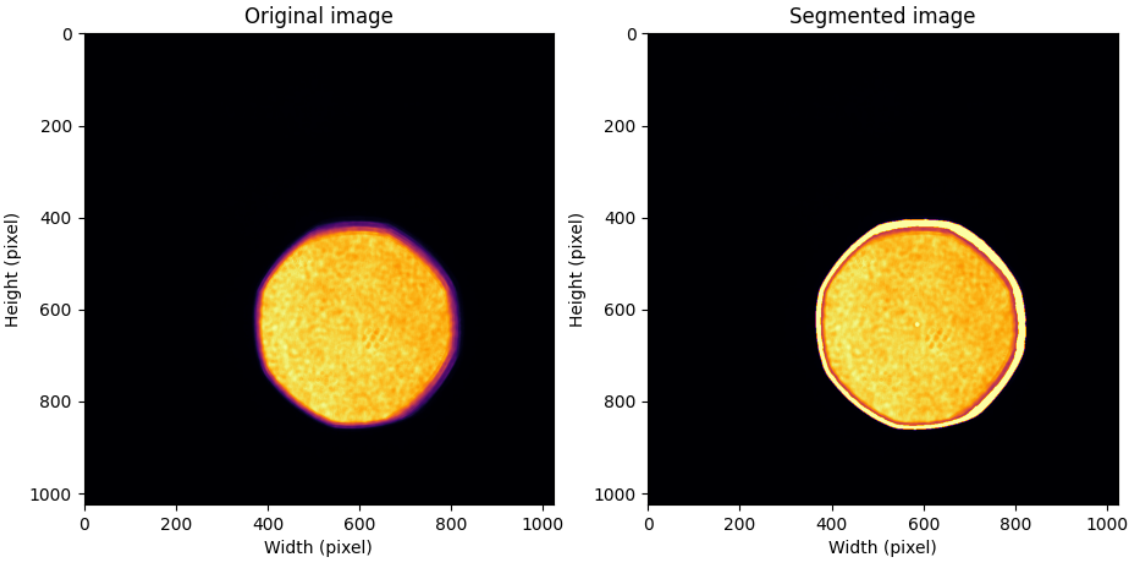

Figure 4.7. *Sample of small spot size and its segmented mask*

given the image contains a laser beam with speckle effect. Thus, controlling the exposure requires successfully running the pipeline in Figure 4.4, prior to calculating optimal camera exposure. This can be problematic and a heuristic is needed for handling edge cases, such as identifying whether the beam is actually present or not, either because the laser is turned off or not pointing towards the camera in the first place. In addition, there might be reflections in the image, due to the multiple filters in place before the camera sensor.

4.8.2 Optimizing CPU usage

After the initial beam profiling pipeline was implemented and it gave acceptable results, further analysis was done on profiling the CPU and memory usage of the beam profiler pipeline itself. For properly profiling and benchmarking the beam profiler, it was necessary to isolate the profiling pipeline algorithm from the rest of the application and execute it on its own. In the test environment, the Python script only transferred raw images from the camera, processed the images and saved the extracted output data. In addition, the script was profiled with both a laptop and the target machine.

For measuring the CPU usage of each function and their internal procedures, a python package `line-profiler` was used. The source code must be modified to mark the functions of interest, whose timing is then measured during execution. An example of taking the measurements is shown in Program 4.3. First, the `@profile` function decorator must be added. Then the application is executed as normal, with the exception of invoking the `kernprof` script. Next the application is executed for sufficiently long time. Since the objective is to identify the most expensive function calls in regards to CPU time, a relatively short test time is sufficient, ranging from 10 to 30 seconds.

```
1  # inject profiling decorators for functions of interest:
2  # In e.g.  application.py:
3  #
4  # @process
5  # def execute():
6  # ...
7  #
8  # Execute, measuring timing
9  python-venv/bin/kernprof -l application.py
10
11 # after execution, generate text-formatted results:
12 python-venv/bin/python -m line_profiler application.py.lprof
```

Program 4.3. *Benchmarking a python script*

Since the beam profiler has an FPS of 1-10, there are already more than 100 executions after 10 seconds. The absolute time an individual function has been executed may not be accurate, but the relative execution times are accurate. After executing the application, execution is stopped and results are saved in the corresponding output file. Then the output is decoded to plain text, which contains actual function and variable names. Invoking the `line-profiler` should not increase the execution time significantly, however, some overhead and an increased execution time is expected.

The initial timings for the beam profiling pipeline are shown in Table 4.2. The measured execution time varied between 10 to 20 seconds, to average out any inconcistencies between each pipeline execution. Total time marks the total execution time of the beam

profiling pipeline. If the pipeline was to execute 100% time and the application was nnot executing the drivers or GUI simultaneously, these would result in FPS of 2.6 for laptop and 0.42 in target machine.

Table 4.2. *Unoptimized beam profiler pipeline execution*

Operation	**Total time spent (µs)**	**Relative time spent (%)**
Laptop (i5-7200U)		
Total time	388810	100
Preprocess	1362	0.3
Smoothing	49177	12.6
Copy image data	966	0.3
Segment image	158257	40.6
Extract output features	180410	46.2
Target machine (Armv7l)		
Total time	2374088	100
Preprocess	6730	0.3
Smoothing	270808	11.4
Copy image data	4141	0.2
Segment image	826297	34.8
Extract output features	1266112	53.3

Table 4.2 shows that, first, the pipeline is faster on a laptop than on the target system, roughly 20 times faster. While this is anticipated, it is not exactly clear which function actually executes faster in the target machine without measuring it. Clearly taking a copy of the data is 4 times slower, where the output extraction is 7 times slower. It seems evident that the extraction slows down the most and thus it is worth being optimized first. In addition, it takes 53% of the total execution time in the target machine, which alone could provide the greatest reduction in execution time, if optimization is possible. Due to the fact that the execution timing differs on machine being used, it is crucial to keep measuring the timings every time when making any adjustments to the beam profiling pipeline. The first options for optimizing the extraction were investigating whether OpenCV or ScienceKit-Image provides faster execution for each function, whether there are any redundant function calls or data copies and whether the execution order has any effect on the timing. As a result, it seems OpenCv executes most of the operations faster. OpenCV always performs faster or does Skimage perform better with certain functions. Data copies are not expensive in terms of CPU-time.

An alternative optimization technique would be to parallelize the execution. Since feature extraction relies completely on segmenting the image, the segmentation cannot

be parallelized from extraction. The extraction, on the other hand, does not mutate the segmented image, and it contains multiple separate function to calculate separate numerical values. Each one of these feature calculations could be parallelized, though. Some of them are expensive, so parallelizing might decrease the execution time.

Based on the measurements presented in Table 4.2, multiple optimization were conducted, always running the benchmark again to verify actual improvements. Notably, an alternative implementations for functions were written, OpenCV and Skimage comparisons were done and unneccessary executions, such as image copies, were eliminated. The optimized results and their timing is presented in Table 4.3. After some novel optimizations, the processing time for single image was reduced by 20% on both a laptop and RaspberryPi system.

Table 4.3. *Optimized beam profiler pipeline benchmark*

Operation	**Total time spent (µs)**	**Relative time spent (%)**
Laptop (i5-7200U)		
Total time	313196	100
Preprocess	1362	0.4
Smoothing	49177	15.7
Copy image data	966	0.3
Segment image	128188	40.9
Extract output features	133503	42.6
Target machine (Armv7l)		
Total time	1882589	100
Preprocess	6730	0.4
Smoothing	270808	14.4
Copy image data	4141	0.2
Segment image	650678	34.5
Extract output features	950232	50.5

4.8.3 Memory usage

Last, the beam profiler's memory usage was analyzed. Although there was no direct necessity for decreasing the memory usage, it is reasonable to verify the memory usage for both short-term and long-term operation. The results were measured with a Python package `memory-profiler` in a similar fashion as the exeuction times were measured in Program 4.3.

Program 4.4 shows capturing and viewing a measurement of a Python script memory usage. The results, shown in Figure 4.8 were measured with a Laptop with 20 GiB of RAM. This affects the interval the Python garbage collector needs to free memory and thus how often garbage collector is executed, which can be considered as an unneccessary execution and might be something worth optimizing.

```
1  # inject profiling decorators for functions of interest:
2  # In e.g.  application.py:
3  #
4  # @process
5  # def execute():
6  # ...
7  #
8  # Execute, measuring timing
9  python-venv/bin/mprof -l application.py
10
11 # plot the memory usage over time
12 # optional argument:  -flame
13 python-venv/bin/mprof plot
```

Program 4.4. *Benchmarking a python script memory usage*

In each execution, about 72 MiB of memory is used for each frame. Also from the figure it is apparent that after a few seconds of warm-up the memory consumption stays well under 500 MiB throughout the execution. The same measurement was measured for a much longer period, 10 minutes, just to ensure that the memory usage in fact stays within th 500 MiB limit.

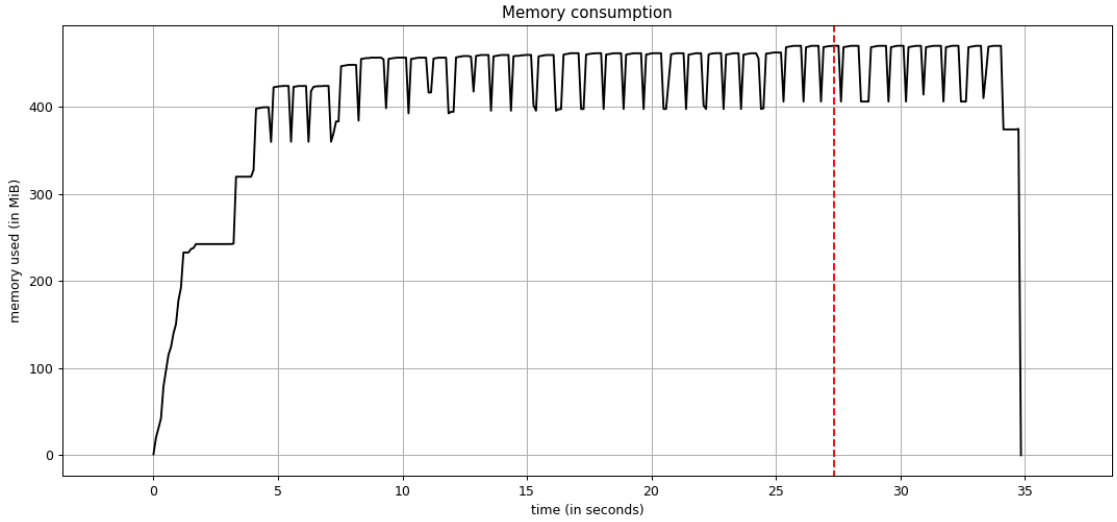

Figure 4.8. *Beam profiler memory usage*

Using a Bash command `cat /proc/{procid}/smap` in Linux it is possible to observe the memory usage of each library during the execution, as seen from the operating system process records. The beam profiler allocates 44 MiB for static libraries, such as Qt and the camera driver. The rest is allocated for heap and is used throughout the execution.

4.8.4 Improving memory usage with mmap

To further optimize the memory useage, the algorithm would need to be revised. Since a single picture is a relatively large array (1024 x 1024 x 8 bit = 1.05 MB), additional copies of the sama data should be made sparingly. Sometimes it is difficult to estimate the copies that external libraries may create. For instance libraries that are used for graphical representation may create additional copies of the data. OpenCV does not copy the array but modifies the input data array directly. While this is beneficial for most of the cases, modifying the original image may render it unusable for other algorithms that are executed concurrently. Since in this project multiple features are extracted separately, each feature requires its own copied data to operate on. So, for instance, extracting 10 features requires creating 10 copies of the image, which in 20 FPS would require allocating 1.05 MB x 10 x 20, requiring 210 MBs memory allocation each second. Also a possible solution would be to allocate a static memory region for use, which can then be memory mapped for use. [38].

Since each feature extraction only partially modifies the original array, it would be possible to use a copy-on-write feature on the underlying data, if the operating system supports it. As this implementation relies on Linux features, a memory mapping can be configured to use copy-on-write mode, as described in the Linux manpage [39]. Depending on the amount of modified data, using the mmap with copy-on-write may save significant amounts of memory that is being actually allocated. Manpage also says

that the original data should not be modified after calling `mmap`, because after modifying the original data the behaviour would be undefined. Numpy array can be memory-mapped using the commands shown in Program 4.5.

```
1 from numpy.lib.format import open_memmap
2
3 # initialize 10 instances of image arrays
4 arrays = open_memmap("/tmp/array", mode="c", shape=(10, 1024,
                                    1024, 3))
```

Program 4.5. *Using a copy-on-write feature with mmap*

While using the memory mapping feature, it is possible to statically allocate all the memory that is required for processing a single image. When statically allocating the memory region, not only is garbage-collector under much less stress to execute, but also it is guaranteed that only only single image's data is stored in memory at a time.

4.9 LIV characterization

LIV characterization is one of the first characterizations to perform for a new device, because many other characterization rely on a known output power from the laser device. The task that the LIV characterization attempts to solve is, as presented in Section 2.1, that the LIV curve is a function between laser current and output power. Although an LIV also describes the relation between the laser current and voltage, it is not of interest here, because in this case the laser modules are driven with constant current and not constant voltage. An LIV characterization is unique to each device and it must be measured individually to determine the system's exact operation.

The characterization procedure is straighforward: sweep the current area that is of interest, measuring power in each step. Smoothing, or averaging, is applied to power measurement for each measurement. Additional values, such as an internal photodiode reading or voltages, is recorded as well. Since the non-lasing region is not of interest, the region is usually swept with far fewer measurement points, whereas the threshold region requires the most measured points. The linear region requires fewer points as well.

The output of the LIV characterization is a lookup table, which maps a laser current to corresponding output power. All other output values are interpolated as needed. For practical reasons, the calibration table cannot have infinite amount of rows, but is limited to about 30 rows. Characterization itself usually incorporates more then 30 measurement points, so the measured points must be fit or mapped into smaller table, preserving the accuracy of the characterization. This is solved with an algorithm that does not select any measured point, but rather interpolates best approximated points. These points are layed out so that the linear area of the LIV curve has fewest points whereas the non-linear threshold area has most of the points in the table. This way, the relative error stays within

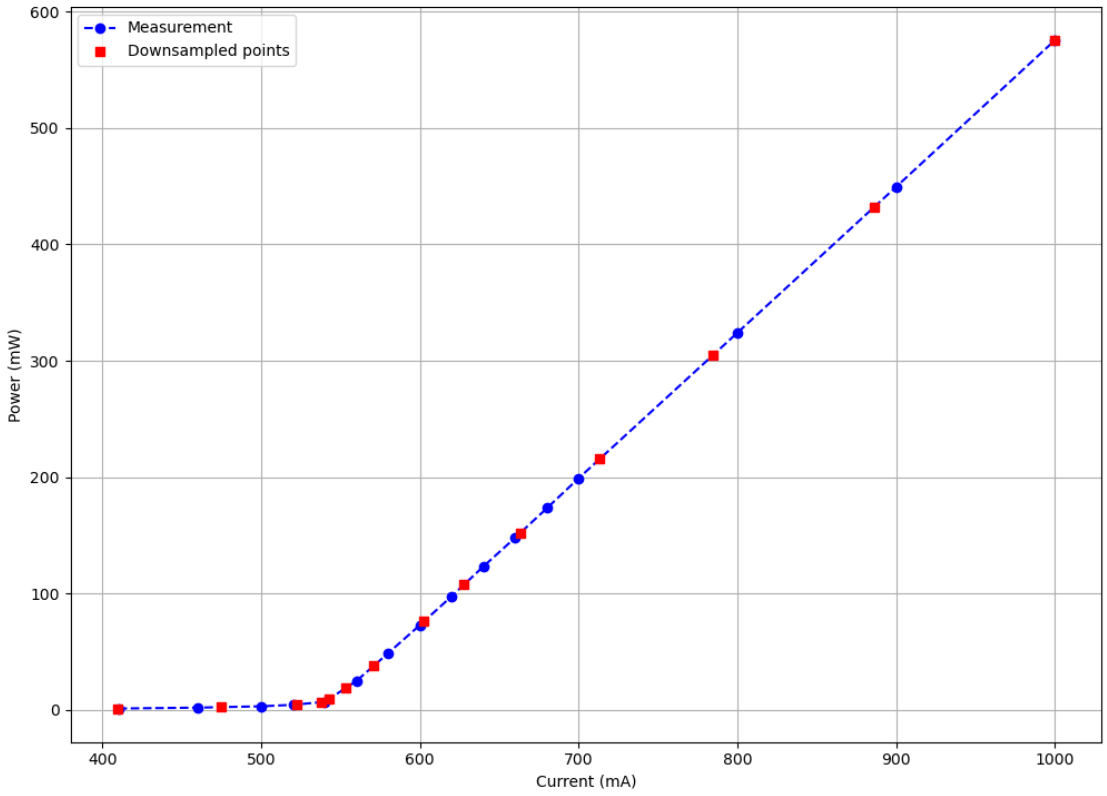

Figure 4.9. *Downsampling LIV characterization*

specified limits. An example of measured and downsampled data is shown in Figure 4.9. A simple way to achieve such table, where most of the points is centered around the thershold area, is by doing a reverse fit for the data. By doing so, the power is mapped to currents, at not the opposite, which is the eventual goal. Since it is known that the threshold is well below power of 10 mW, roughly half of the data points can be fixed to powers of 0-10mW. The other half of the points can be distributes along the two linear areas outside thershold area.

The characterization is validated against the ideal LIV curve for known laser characteristics. First, the linear area must be linear all the way to the maximum output power with small relative error allowed. Second, the output power must reach a required minimal limit. Third, the curve must be non-monotonic for interpolation to work correctly. Usually non-monotonic values don't occur in the linear region, since the relation between the laser current and output power is in fact monotonic. But in the non-lasing region, where output powers are small and noise level is relatively high, the power measurements may have non-monotonic values. Non-monotonic values are either discarded, or if there are too many values to be discarded, the device must be characterized again.

The laser's temperature must also be monitored during the LIV characterization to retain the characterization accuracy. Drifting the temperature causes not only the wavelength but also the laser's output power to drift as well, as described in 2.1. In this

measurement setup, both the wavelength and power affect the measured power, because the power meter is calibrated for the specific wavelength of the target system. In the case of a temperature drift, given the device controls temperature correctly, it is enough to pause the characterization process for a few seconds and try again, letting the device temperature stabilize first. Although the laser device monitors and controls laser module's temperature, during abnormally fast power sweeps, which do not occur in normal treatment situations, it is possible to drift the temperature for short periods of time. Certain edge cases may occur, which must be handled and the characterization may even need to be run again. LIV graph also verifies that the laser device operates as expected and is not a faulty unit. For a faulty unit a closer inspection is needed, ranging from properly securing connectors to replacing the laser modules or some other components.

4.10 Chacterizing calibration system's internal parameters

As the calibrator system itself contains electrical and optical subsystems, the various subsystems need to be characterized as well. For instance, the built-in power meters need to be characterized for the intended use. A power meter in this case is a photodiode-based current source, which is first converted to voltage and amplified, and after that it is converted to digital signal with an ADC and transferred with SPI bus the PC. So both the voltage conversion and the amplification response needs to be determined for correct gain and offset.

The photodiode's current response is linear to the light it receives, so only a linear fit, consisting of gain and offset combination, should be enough for characterizing the power meter. However, for an extremely low powers the linear fit doesn't may produce high relative errors, even if the absolute error stays low. Fox decreasing the relative error, a second linear fit was made in low-power region. These two fits together provide an excellent relative and absolute error throughout the operating area.

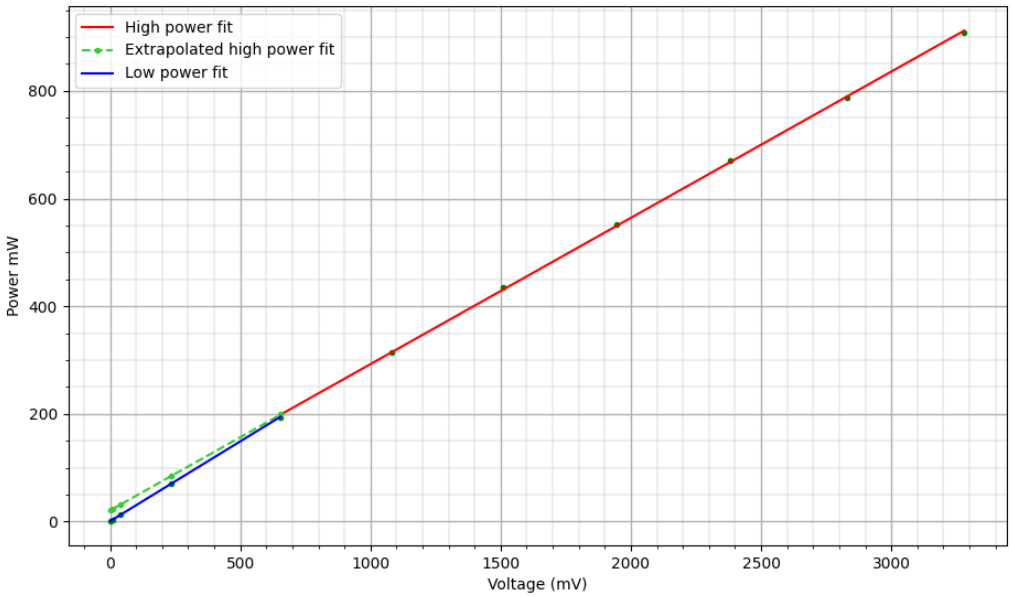

Figure 4.10. *Power meter characterization*

Generating two fits is essentially an optimization problem, where both relative and absolute errors across the regions are the parameters to be minimized. A sample characterization is shown in Figure 4.10, where two fits are used for power meter characterization. In this scenario, the first few samples are used for the lower fit, and the rest of the data points are used for the upper fit. Upper fit is then etrapolated to lower powers to visualize the difference between the lower and the upper fit in lower power region.

The most important parameter to control during the characterization is the cutoff power between the high and low power fits. It is even possible that one fit provides good enough results. A python package `scipy` provides function `optimize` for this kind of optimization tasks, where a parameter is algorithmically and iteratively tuned while the output parameter is being minimzed. The only thing left to do is to setup the optimization problem correctly for `scipy.optimize`. While the power meter's characterization should be constant over time, it is considered a best practice to periodally check and correct the characterization. Thus, the power meters need to be characterized yearly to guarantee their operation.

5 VALIDATION

Since the device that is being calibrated is a medical device, the tools used to manufacture and calibrate one must be validated as well. In the context of this work it has the implication that the calibration system itself must be verified and validated. Verification protocol ensures that the specification describes the operation of the calibration system. After verification, a validation protocol is executed according to a validation plan by personnel not affiliated with construction of the calibration system. The validation protocol is a series of individual tests that one can run with the required equipment. Each test verifies a single feature or operation, and usually multiple tests are required for verifying a single specification item. A protocol test contains detailed instructions with ideally zero ambiguity, along with a defitinion for pass criteria, which has to be met in order for the test to pass.

Only after each test in the validation protocol has passed, the calibration system can be used in a production environment for calibrating medical laser systems prior to shipping them to customers. Every time the calibration system is modified in any way, the complete verification and validation protocol must be executed again prior to re-entering the production phase. After calibrating a device, it must be verified with meters that were not being used during the calibration. This is to eliminate the effect of faulty meters during the calibation.

A detailed validation protocol was created to verify the calibration system's operation, containing multiple dozens of tests. A sample validation protocol is presented in Table 5.1. Since the system contains a dozen calibrations, each calibration procedure was validated separately. All of the customized electronics and meters were validated against a commercial, comparable meters, such as a power meter or a spectrometer. All digital processing must be validated as well, in the level that the results must be consistent and the medical device must still operate as specified.

Table 5.1. *A sample of the validation protocol*

	Requirement	**Validation method**	**Acceptance criteria**
1.	Laser system is fully operable after automatic calibration	Validate device according to device verification sheet	Device passes all verification tests
2.	Calibration data and raw measurements are found in external storage	Check external storage it has all data related to calibration	Calibration data exists on external storage
3.	laser device is interoperable with any optical beam shaper unit	Operate evice with other optical beam shaper units	Laser system operates as specified in the product specifaction
4.	Capable of measuring laser device's internal reference voltages	1. Characterize vref as per process instructions. 2. Measure vref manually using a multimeter.	Vrefs measured with calibation system and a multimeter match each other.
5.	Software can read and write device configuration files from file system	Read and write configuration file to local file system	Configuration file is intact
6.	Software can read and write device configuration files from target device	Read and write configuration file contents to target device	Device configuration is identical after writing and reading it.
7.	Software can read and write device configuration files to the external storage	Read and write configuration file contents to the external storage	Device configuration is identical after writing it to the external storage.

5.1 Evaluating the calibration quality

To evaluate the performance and the accuracy of the overall device calibration, a total of 20 devices were being calibrated and their operation was verified using external beam profiler and power meters. The results are shown in the following sections. Overall, the evaluation was sucessful, since each device fulfills the medical device's s'specification, guaranteeing that the device operates properly.

5.1.1 Beam profiler

The beam profiler is validated against a commercial profiler. Although the algorithms differ to some extent, and commercial one is a black box in terms of its internals, the results should be comparable, especially when the beam is circular. In some cases the profiler that was being built in this project has better accuracy than a comparable commercial beam profiler due to its different, simpler algorithm, that is being used for determining the beam spot area.

Spot size calibration was evaluated verification, three separate spot sizes for each unit. The results are shown in Figure 5.1. A spot size target was given to the device, for instance 1000 µm. The actual spot size was then measured with an external beam profiler and the spot size diameter was recorded. It must be noted, that since any digital beam profiler relies heavily on purpose-built algorithms, the results may be different depending on the beam profiler. It is known that the external beam profiler measures beam diameter differently than the one built during this project.

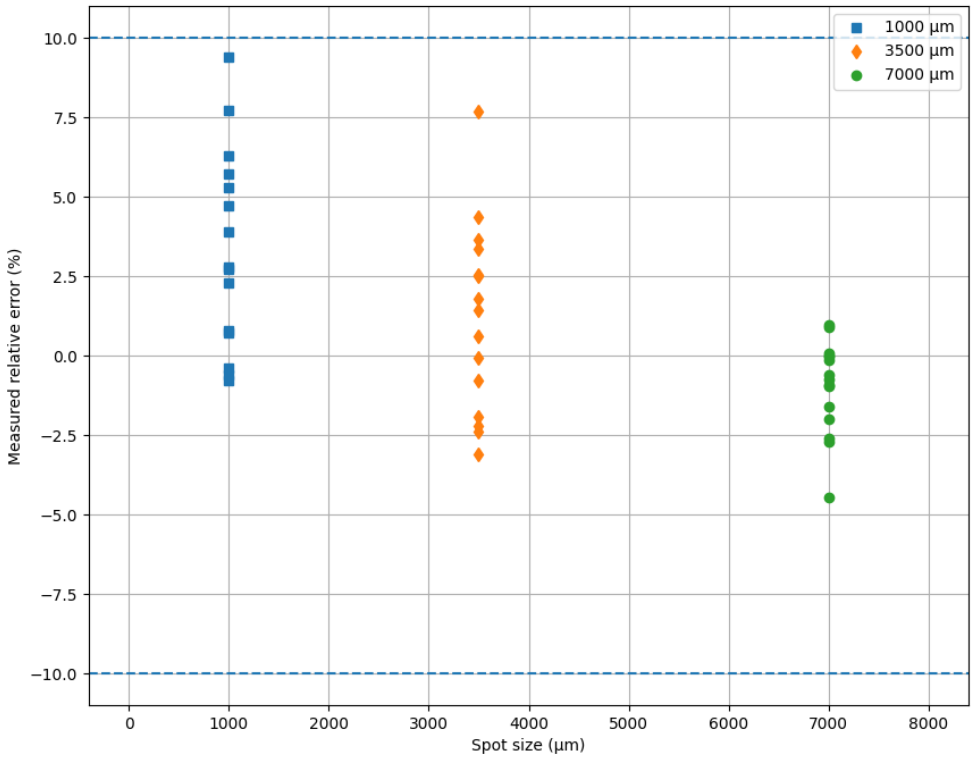

Figure 5.1. *Spot size measurements*

From the Figure 5.1 can be seen that the implemented beam profiler seems to overestimate the size of the smaller spot sizes, since the lowest spot sizes of 1000 µm are generally higher than reference. Also smaller spot sizes have more variance and have greater inaccuracy in the calibration process itself. This is likely due to the fact that

smaller spot sizes are not perfectly circular and have minor imperfections, which can be depicted in Figure 4.7. Since the implemented beam profiler calculates the average diameter using the whole beam spot area rather than just two diagonals, the diameter can differ from the external beam profiler. Figures 5.2 and 5.3 show no outliers, indicating that the characterization setup works as expected.

Table 5.2. *Relative error for beam profiling*

Spot diameter	Mean relative error	Standard deviation	Absolute maximum relative error
1.0 mm	4.0 %	3.8	9.4 %
3.5 mm	1.1 %	2.9	7.7 %
7.0 mm	-1.0 %	1.4	4.5 %

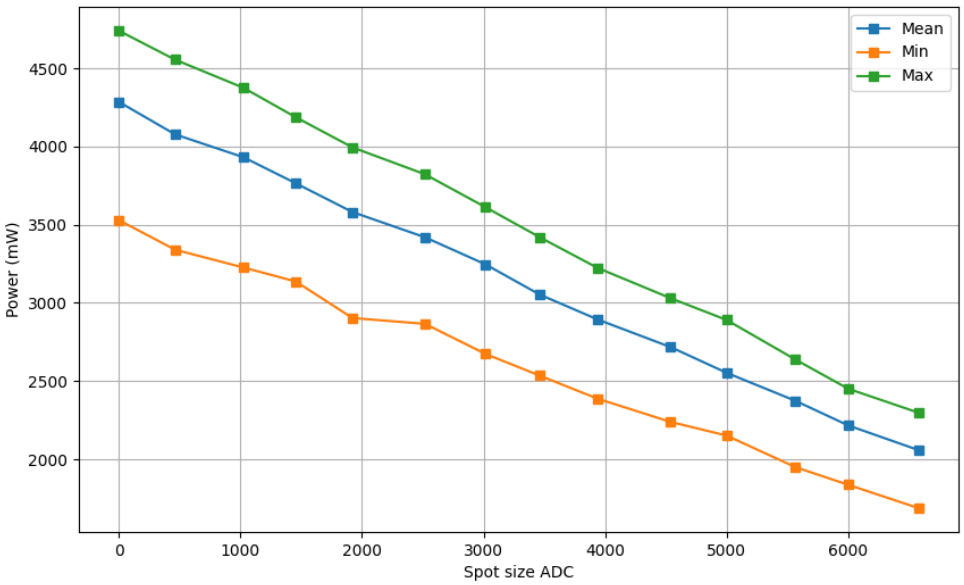

Figure 5.2. *Spot size ADC 1 channel*

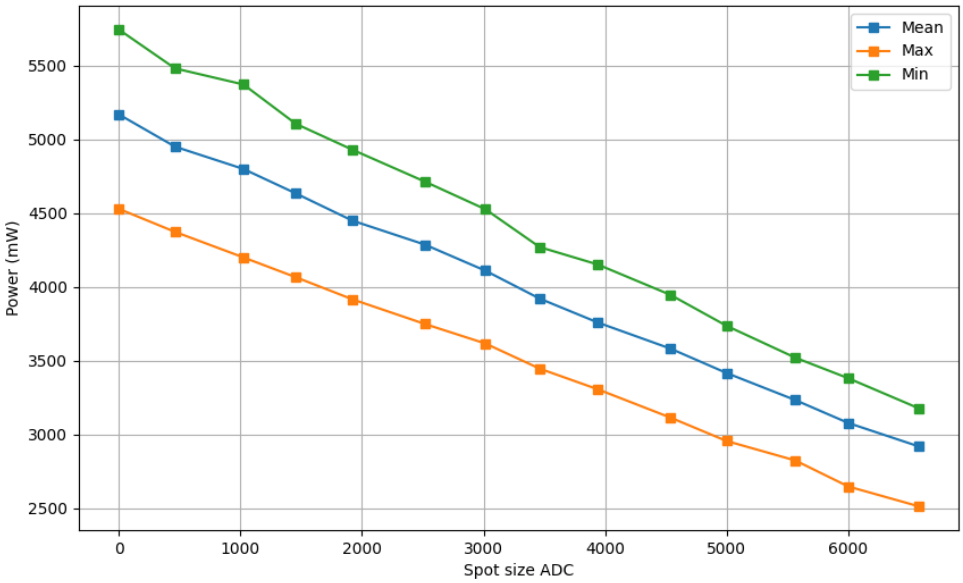

Figure 5.3. *Spot size ADC 2 channel*

5.1.2 LIV-characterization

LIV-characterization, in which the laser's optical power is measured against the laser's driving current, requires multiple steps for full validation. First, an operator must run the full characterization procedure, following corresponding process instructions. Next the in-house built power meters are validated and verified using a commercial and calibrated power meter and results are compared. Then the LIV-characterization is executed manually with external power meter and then the characterization results are compared. At this point, it is known that not only are the measurements accurate, but the post-processing is also accurate and gives correct results. Last, the characterization data is saved to the device as well as to an external storage and both are read and their content is verified, to ensure that the data is identical with the original output data. Since there are effectively four separate power meters in the setup, each one being used for different optical outputs and wavelengths, they must be verified separately.

All 20 calibrated devices were verified using an external power meter. For each device, three power levels were being tested. As depicted in Figure 5.4 and Figure 5.5, calibrated devices fulfill the device specification. With the treatment beam, Table 5.3 shows that calibrated mean error grows linear with power, indicating that the caliration system's internal calibration is non ideal. As for the aiming beam, there is no similar linear trend with relative mean errors, as seen in Table 5.4.

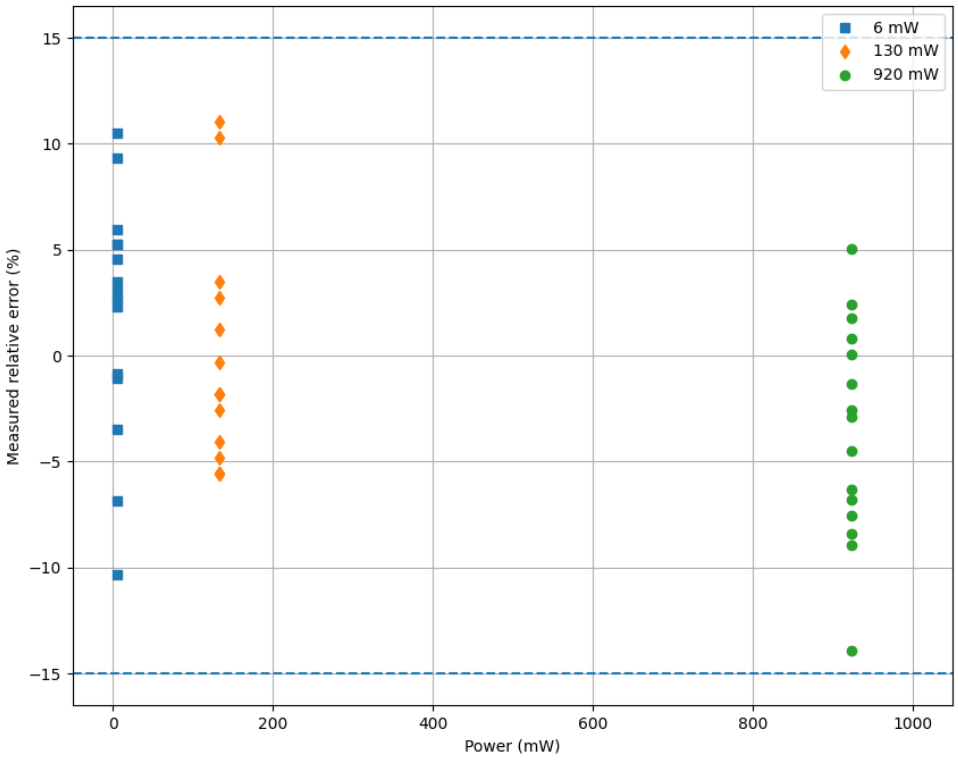

Figure 5.4. *Treatment beam LIV verification*

Table 5.3. *Relative error for treatment beam LIV*

Power	Mean relative error	Standard deviation	Absolute maximum relative error
6 mW	2.0 %	5.5	10.5 %
130 mW	0.4 %	5.1	11.0 %
920 mW	-3.5 %	5.0	13.9 %

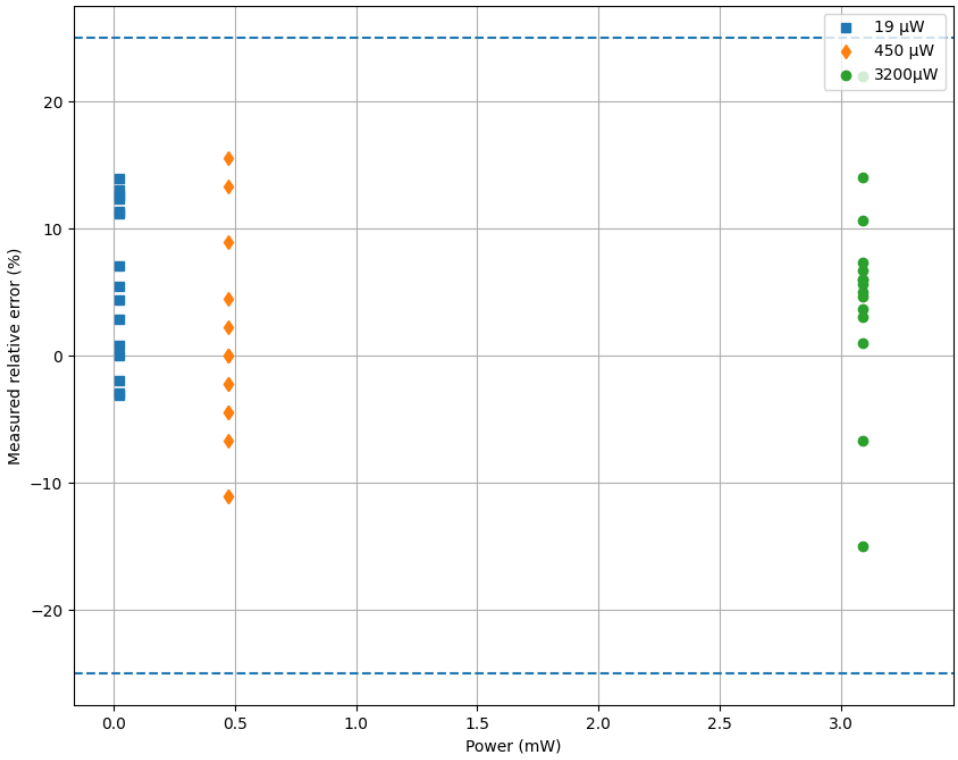

Figure 5.5. *Aiming beam LIV verification*

Table 5.4. *Relative error for aiming beam LIV*

Power	Mean relative error	Standard deviation	Absolute maximum relative error
19 µW	4.6 %	7.4	12.6 %
450 µW	0.1 %	7.6	15.6 %
3200 µW	4.8 %	8.0	22.0 %

5.2 Calibration system

The calibration system must have a capability to communicate with the device as well as read and write configurations from it, as was specified. JSON (JavaScript Object Notation) is being used as the serialization format in PC, but any other format, such as plain text or yaml, could have been selected as well. The calibration system must be able to produce and serialize the json file from the device's configuration, as well as upload the complete deserialized JSON file to the device's memory. In addition, these JSON files are persisted to an external database in a text format. The validation simply tests that uploading and downloading from and to both the device and the external storage works and file content stays intact. The device also has checksums in place for detecting and

preventing corrupted configurations, so verifying the communication with the device is not necessary in this context.

The user must be able to view certain device parameters, such as temperatures, controlling parameters and some intermediate control values. The existence for each required value is verified separately with a test for each parameter. Some of these are user-editable, but most of them are read-only.

5.3 Laser system interoperability

The medical system consists of two components: a laser device system and an optical beam shaper unit. A requirement for the calibration system is to enable the evice to use any optical beam shaper unit and still produce output power with high accuracy, as specified in the system's requirements. One of the motivations for an automated and comprehensive characterization was to enable the interoperability between different optical optical beam shaper units, without having to characterize each set of laser devices and optical beam shaper units with each other.

For a complete interoperability, several parameters need to characterized for both to the laser device and to the optical beam shaper unit. With a successful characterization process, replacing the beam-shaper-unit should still result in the medical system to operate under it's specification.

6 IMPROVEMENTS

Although the project requirements were met during the development of the calibration system, the software components that were implemented are planned to be used in other projects as well. Having a modular design and implementation, it is easy to modify the application and extract components and functionality as is needed. There are still multiple options for refining and tuning the application or its components. In this chapter there are listed a few scenarios, which might improve the performance of either the whole calibration system, or some components or measurement equipment. It is possible that in the future some of these improvements will be investigated further or implemented as needed.

6.1 ADC noise filtering

When converting analog signals to digital, both the measured signal and the conversion introduce some noise into the output. The input signal will inherently contain background noise and may additionally contain some electrically induced noise from the printed circuit board and components next to the chip. In addition, the ADC conversion introduces various noises, such as quantization noise [25].

One possible mitigation for decreasing the noise level incorporates digital signal processing methods. In the case of unwanted noise, a digital low-pass filter would be suitable, which would only allow frequencies of interest, such as < 5 Hz, to pass through. Such a DSP filter is unfortunately not effective with the current system, because the ADC sampling interval is not deterministic and inherently contains some jitter. This is due to the fact that the sampling synchronization is done on top of an operating system. A novel implementation would use a moving average filter, decreasing the noise level by some degree. A more effective solution, however, would be to incorporate a microntroller and delegate the ADC sampling to it and transfer buffered values with e.g. SPI bus the host PC. An alternative to this would be to write a Linux kernel driver for the ADC and possibly incorporate a Real-Time-Kernel. But the problem with the kernel module is that it would require a significant amount of work and it would be completely tied not only to the operating system, but also to the architecture at hand. Also maintaining a kernel driver significantly increases the time and resources needed for maintaining the system.

6.2 Accelerating beam-profiling

Beam profiling is algorithmically the most expensive operation in this calibration setup. The reason is simply that an image of 2000 x 2000 pixels requires a significant amount of arithmetic operations even for simple transformations. With high enough FPS and a complicated pipeline this will easily become a bottleneck, which will limit the software's capabilities. An easy way to decrease the stress, especially in high-FPS scenarios, is by downsampling the image to a smaller size, for example to 1/4th or even 1/16th of the original image. Decreasing the resolution may, however, decrease the quality of the measurement and may introduce an additional noise to the image, decreasing the profiler's performance.

Python 3 provides various libraries and interfaces, such as OpenCv and SciKit-Image, that are used for image manipulation. Both of these use Numpy-arrays as the underlying data structure, where indexing and data manipulation is fairly performant. However, given that the calibrator has to keep up with other tasks as well, such as frequently communicate with the devices, update GUI and run the calibration state machine itself, it cannot afford to use 100% of the CPU time only for the image manipulation.

Usually this is resolved by splitting the program into multiple threads, which can then be run with multiple CPU cores in parallel. But, in case of Python language, the Python Global Interpreter Lock (GIL) prevents the use of multithreaded computing, effectively rendering multithreaded applications to a single thread at a time [40]. To overcome this limitation, there are a couple of solutions. First, Python has a module library called `multiprocessing` for running multiple processes, not just threads. Separate processes are not limited by GIL, enabling true parallel computing between processes. While multiprocesing would theoretically solve the problem at hand, it introduces other issues, mainly the problem of sharing data between processes, which is an error-prone and difficult task to do.

Sharing the program's state between multiple processses is discouraged, and a queue-based message passing is recommended instead. Message sharing works well, when the data that is being sent is fairly modest in size. Normal Python objects will be easy to pass between processes. But sharing images of multiple Megabytes will likely result in not only additional copies of the data, it may even be challenging to share that data directly. An alternative may be by using memory-mapping with `mmap` syscall, which will allocate shared memory area, which multiple processes can then access and manipulate.

An second alternative to threading in Python would be to offload the image manipulation, or parts of it, to a separate library or binary. Such library could written with lower level language that has been optimized for speed, for instance with C++. The library would export an interface for the image manipulation process or separate functions for parts of it, then execute the processing in a separate thread. Calling the interface from Python wouldn't then consume the Python process's time, since the processing would be done in a separate thread. This way, the python threads would be available to execute other

operations simultaneously. The difficulty in creating an external library is to complexity it adds. The library needs to be written and built for multiple operating systems, because the profiler needs to cross-platform program as well.

Another alternative for improving the beam profiler performance would be to parallelize the algorithm itself. Table 4.3 shows that, on target machine, the output extraction is the most CPU intensive operation. While the segmentation isn't as easy to parallelize, because it is a sequential operation, constantly mutating the data, the feature extraction phase does not mutate the data in any way. Thus it would be possible to run each distinct extraction in separate threads, vastly decreasing the time spent on extraction. The segmentation could be partly parallelized as well, with the cost of added complexity. Since image data is not modified, the only overhead would be to start threads and and wait for each one of them. For an improved results, a threadpool of background tasks would likely be the best option.

Additionally, a hardware acceleration would provide substantial improvements [24]. For instance, OpenCV has a limited support for accelerating the computing with external resources, such as a GPU. But the acceleration once again adds much more complexity to the system so the benefit must be significant and it is likely the last option to try. Although RaspberryPi includes a GPU, acceleration would most likely require additional hardware or a GPU that supports certain matrix operations directly.

6.3 Generalizing the beam profiling pipeline

It is likely that the laser beam profiling software, that has been built during this project, will be used in other projects as well. In current implementation, the beam profiling pipeline has been optimized to work with the laser system at hand. Although it works as expected, multiple components, such as the feature extraction and segmentation, can be used in other projcets as well. On the other hand, it is likely that exactly these two will be modified to suit other projects. Through OOP pattern, inheritance, and clear separation of responsibility, it would be possible to create highly modular implementation for beam profiling.

Depending on application and optics the laser looks a bit different in the image. Due to the nature of the divergence of the laser beam it has either gaussian intensity curve, or it is sharp edged beam. For a complete abstraction and modularity, the camera driver, along with each step in the beam profiling pipeline should be independent and completely opt-out, allowing any project to use the part of the pipeline that is needed.

6.4 Possibility for remote calibration

One possible further development task for this calibration system is the possibility of calibrating devices remotely. The rationale behind this scenario is that, as the products reside around the globe, it is costly and creates downtime to ship devices back to the

factory for calibration. The option for remote calibration has been taken into account in the calibration system's requirements.

With a remote calibration capability it would be possible to calibrate the devices either in certain remote locations that have the capability to calibrate, or to ship the calibration system directly to the client. Most of the requirements related to the remote calibration capability arise from guaranteeing a safe usage even for non-trained personnel, as well as for enforcing correct setup of instruments for each calibration phase.

7 CONCLUSIONS

The objective of this thesis was to develop a working software in the calibration system, for calibrating a medical laser system. The target of calibration, the medical laser system, consists of three central components: a laser device, an optical beam shaper and an external graphical user interface. The calibration system includes a multitude of both commercial and custom built equipment, such as power meters, spectrometer and a camera.

The intent was to create a software that utilizes various electronics and equipment, which is capable of calibrating the medical system as a whole, so that the device operates as it is specified. In order to characterize the laser system, multiple separate characterizations are required, some of which are fully automatic and some of which require user interaction.

When developing the calibration software, close interaction with hardware and multiple physical peripherals for communicating with the equipment was required. After that, all the calibration procedures were written, often consisting of various post processing steps. Algorithmically, the most complex piece was the beam profiler, which incorporated image processing and various machine vision techniques. The beam profiler is comparable to the performance of a beam profiler software available on the market.

During the development process multiple refactorings were required. While it may give the impression that the initial implementation was of poor quality, it is actually the nature of software development that the program is in fact evolutionary and iteratively built. After writing each software component one by one, the complexity increased slowly. After some time, it made sense to decrease the complexity by refactoring the software components.

The results were very promising, since an assembled device can be verified to operate as specified after it has been succesfully calibrated with the developed system and software. After validating the calibration software and the system around it, it can be concluded that the calibration system is suitable to be used in a production of the medical laser system.

REFERENCES

[1] Bhaskarabhatla, A. and Klepper, S. Latent submarket dynamics and industry evolution: lessons from the US laser industry. eng. *Industrial and corporate change* 23.6 (2014), 1381–1415. ISSN: 0960-6491.

[2] Belforte, D. A. A Great Year for the Industrial Laser Business in the USA. eng ; ger. *Laser-Technik-Journal* 15.2 (2018), 30–31. ISSN: 1613-7728.

[3] Paasch, U. The Future of Fractional Lasers. *Facial Plast Surg* 32 (2016).

[4] Paasch, U. and Grunewald, S. Update on dermatologic laser therapy II – advances in photodynamic therapy using laser-assisted drug delivery. eng. *Journal der Deutschen Dermatologischen Gesellschaft* 18.12 (2020), 1370–1377. ISSN: 1610-0379.

[5] Kharkwal, G. B., Sharma, S. K., Huang, Y.-Y., Dai, T. and Hamblin, M. R. Photodynamic therapy for infections: Clinical applications. eng. *Lasers in surgery and medicine* 43.7 (2011), 755–767. ISSN: 0196-8092.

[6] Sroka, R., Stepp, H., Hennig, G., Brittenham, G. M., Rühm, A. and Lilge, L. Medical laser application: translation into the clinics. eng. *Journal of biomedical optics* 20.6 (2015), 061110–061110. ISSN: 1083-3668.

[7] Kwiatkowski, S., Knap, B., Przystupski, D., Saczko, J., Kędzierska, E., Knap-Czop, K., Kotlińska, J., Michel, O., Kotowski, K. and Kulbacka, J. Photodynamic therapy – mechanisms, photosensitizers and combinations. eng. *Biomedicine & pharmacotherapy* 106 (2018), 1098–1107. ISSN: 0753-3322.

[8] Caltagirone, L., Bellone, M., Svensson, L. and Wahde, M. LIDAR–camera fusion for road detection using fully convolutional neural networks. eng. *Robotics and autonomous systems* 111 (2019), 125–131. ISSN: 0921-8890.

[9] Klotzkin, D. J. *Introduction to Semiconductor Lasers for Optical Communications: An Applied Approach*. eng. Cham: Springer International Publishing AG, 2020. ISBN: 9783030245009.

[10] *Semiconductor lasers : fundamentals and applications*. eng. Woodhead Publishing series in electronic and optical materials, number 33. Cambridge, UK: Woodhead Publishing Limited, 2013. ISBN: 0-85709-640-0.

[11] *FDA: How to Determine if Your Product is a Medical Device. FDA instructions for identifying medical device*. Mar. 19, 2021. URL: https://www.fda.gov/medical-devices/classify-your-medical-device/how-determine-if-your-product-medical-device (visited on 03/19/2021).

[12] *FDA: Classify Your Medical Device. FDA instructions for classifying medical devices*. Mar. 19, 2021. URL: https://www.fda.gov/medical-devices/overview-device-regulation/classify-your-medical-device (visited on 03/19/2021).

[13] Drugs, Devices, and the FDA: Part 2: An Overview of Approval Processes: FDA Approval of Medical Devices. eng. *JACC. Basic to translational science* 1.4 (2016), 277–287. ISSN: 2452-302X.

[14] *FDA Premarket notification*. Mar. 19, 2021. URL: https://www.fda.gov/medical-devices/premarket-submissions/premarket-notification-510k (visited on 03/19/2021).

[15] *Linux Spidev userspace API documentation. Version 5.11.0-rc4*. Jan. 20, 2021. URL: https://www.kernel.org/doc/html/latest/_sources/spi/spidev.rst.txt (visited on 01/20/2021).

[16] Madieu, J. *Linux deivce drivers development: develop customized drivers for embedded Linux*. eng. 1st ed. Birmingham, UK: PACKT Publishing, 2017. ISBN: 1785280007.

[17] *History and License*. Nov. 17, 2021. URL: https://docs.python.org/3/license.html (visited on 11/17/2021).

[18] *Python 3.9 Documentation*. Feb. 26, 2021. URL: https://docs.python.org/3/ (visited on 02/26/2021).

[19] F. Lott, S. *Functional Python programming: discover the power of functional programming, generator functions, lazy evaluation, the built-in itertools library, and monads*. eng. Packt Publishing, 2018. ISBN: 9781788621854.

[20] *Design of CPython's Garbage Collector*. Oct. 28, 2021. URL: https://devguide.python.org/garbage_collector/ (visited on 10/28/2021).

[21] *Qt 5.12 Documentation*. Nov. 17, 2021. URL: https://doc.qt.io/qt-5.12/ (visited on 11/17/2021).

[22] Krajewski, M. *Hands-On High Performance Programming with Qt 5: Build Cross-Platform Applications Using Concurrency, Parallel Programming, and Memory Management*. eng. Birmingham: Packt Publishing, Limited, 2019. ISBN: 9781789531244.

[23] *PyQt5 Things to be Aware Of: Garbage Collection*. Oct. 31, 2021. URL: https://www.riverbankcomputing.com/static/Docs/PyQt5/gotchas.html#garbage-collection (visited on 10/31/2021).

[24] Forbes, E. *Learning concurrency in Python: speed up your Python code with clean, readable, and advanced concurrency techniques*. eng. 1st ed. Birmingham: PACKT Publishing, 2017. ISBN: 178728316X.

[25] Carvalho, J. M. d. *Digital signal processing*. eng. New York: Momentum Press, LLC, 2019. ISBN: 1-947083-91-0.

[26] Déniz, O., Bueno, G., Salido, J. and De la Torre, F. Face recognition using Histograms of Oriented Gradients. eng. *Pattern recognition letters* 32.12 (2011), 1598–1603. ISSN: 0167-8655.

[27] Klare, B. F. and Jain, A. K. Heterogeneous Face Recognition Using Kernel Prototype Similarities. eng. *IEEE transactions on pattern analysis and machine intelligence* 35.6 (2013), 1410–1422. ISSN: 0162-8828.

[28] *Deep learning in computer vision : principles and applications*. eng. First edition. Digital imaging and computer vision. Boca Raton, FL: CRC Press/Taylor and Francis, 2020. ISBN: 1-351-00380-1.

[29] Nixon, M. S. *Feature extraction & image processing for computer vision*. eng. 3rd ed. Oxford: Academic, 2012. ISBN: 0-12-397824-6.

[30] Soenksen, L. R. and Yazdi, Y. Stage-gate process for life sciences and medical innovation investment. *Technovation* 62-63 (2017), 14–21. ISSN: 0166-4972. DOI: 10.1016/j.technovation.2017.03.003. URL: https://www.sciencedirect.com/science/article/pii/S0166497216302474.

[31] *Asyncio library documentation. Python 3.8 documentation, asyncio library*. Apr. 9, 2021. URL: https://docs.python.org/3.8/library/asyncio.html#module-asyncio (visited on 04/09/2021).

[32] *RPi.GPIO Python package for GPIO in RaspberryPi*. Apr. 23, 2021. URL: https://sourceforge.net/projects/raspberry-gpio-python/ (visited on 04/23/2021).

[33] *RaspberryPi configuration documentation*. Sept. 1, 2021. URL: https://www.raspberrypi.org/documentation/computers/configuration.html (visited on 09/25/2021).

[34] Parker, M. *Digital video processing for engineers a foundation for embedded systems design*. eng. Amsterdam, 2013.

[35] *SciKit-Image Filtering methods reference*. Apr. 16, 2021. URL: https://scikit-image.org/docs/0.18.x/api/skimage.filters.html (visited on 04/16/2021).

[36] Davies, E. R. *Computer and machine vision theory, algorithms, practicalities*. eng. Amsterdam, Netherlands, 2012.

[37] *Laser beam shaping applications*. eng. Second edition. Optical Science and Engineering. Boca Raton, Fla: CRC Press, 2017 - 2017. ISBN: 1-5231-1811-3.

[38] *Numpy documentation for using mmap*. Aug. 3, 2021. URL: https://numpy.org/doc/stable/reference/generated/numpy.memmap.html (visited on 08/03/2021).

[39] *Linux manpage for mmap syscall*. Aug. 3, 2021. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (visited on 08/03/2021).

[40] *Thread State and the Global Interpreter Lock. Python 3.8 documentation, Python/C Api Reference Manual*. Feb. 26, 2021. URL: https://docs.python.org/3.8/c-api/init.html#thread-state-and-the-global-interpreter-lock (visited on 02/26/2021).