VISION TRACKING SYSTEM FOR IN-SITU TISSUE BIOPRINTING

KATERYNA ZAKHAROVA

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN MECHANICAL ENGINEERING

YORK UNIVERSITY

TORONTO, ONTARIO

AUGUST 2024

ABSTRACT

Maintaining printing accuracy despite the constant motion is one of the main difficulties in printing on moving surfaces. Traditional 3D printers are made for stationary platforms, so changing them to work on dynamic surfaces calls for the creation of new control algorithms and methodologies.

In situ (from Latin: in its original place) printing refers to a process where bioprinting or additive manufacturing technologies are used directly at the site of application. The core component of our vision tracking system is a real-time algorithm powered by a Deep Learning model that monitors the printing surface – a harmed part of the human body. This system relies on advanced Computer Vision technologies to accurately track the moving surface's position and orientation.

Our experiments in simulations proved the concept of the robot's ability to adjust to a dynamic environment. A multi-camera system and a 6-DOF robot arm are included in the experimental setup of this research.

Keywords: robotics, computer vision, deep learning, video processing, 3D printing on moving surfaces, 3D bioprinting, quality assurance, real-time analysis.

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my supervisor, Dr Alexander Czekanski, for such an incredible opportunity. As well as for his experience and support throughout my research journey. His invaluable guidance was instrumental in bringing this project to completion.

I am indebted to my lab mates. Special thanks to Dr Salman for his crucial advice, feedback, and expertise, which supported my research project. Usman, Ragab, Dr Daphene, and Youssef, thank you for your camaraderie and assistance in these challenging times for me.

Also, I cannot refrain from mentioning all the people and volunteers who showed solidarity and helped newcomer Ukrainians settle down here in Canada.

Last but not least, I would like to recognise my partner, Maksym. His unwavering support, patience, love, and belief gave me the strength and motivation to persevere. And special thanks to my mom, who taught me not to stop whatever is happening in life.

Thank you all.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS AND ABBREVIATIONS

AM – Additive Manufacturing

CNN – Convolutional Neural Network

CV – Computer Vision

DC – Dice Coefficient

DL – Deep Learning

DOF – Degrees-of-Freedom

FIFO – First In, First Out

FPS – Frames Per Second

MCP – Multi-Camera Photography

ML – Machine Learning

ROI – Region of Interest

ROS – Robotic Operating System

SIFT – Scale Invariant Feature Transform

1 Introduction and Justification

1.1 3D Bioprinting

Back in 2002, Thomas Boland and his colleagues at Clemson University achieved a spectacular feat when they used a modified inkjet printer to print the first live cells (Fig. 1.1). They proved the ability to manufacture live tissues and organs using 3D printing technology by demonstrating the printed cells, which survived for several weeks [1]. The discovery to create functioning biological structures that may be utilised for study, drug testing, or transplantation has opened new opportunities for enhancing the healthcare field. Maintaining cell viability, vascularisation, and structural integrity of the printed tissues are just a few of the difficulties the further research must overcome.

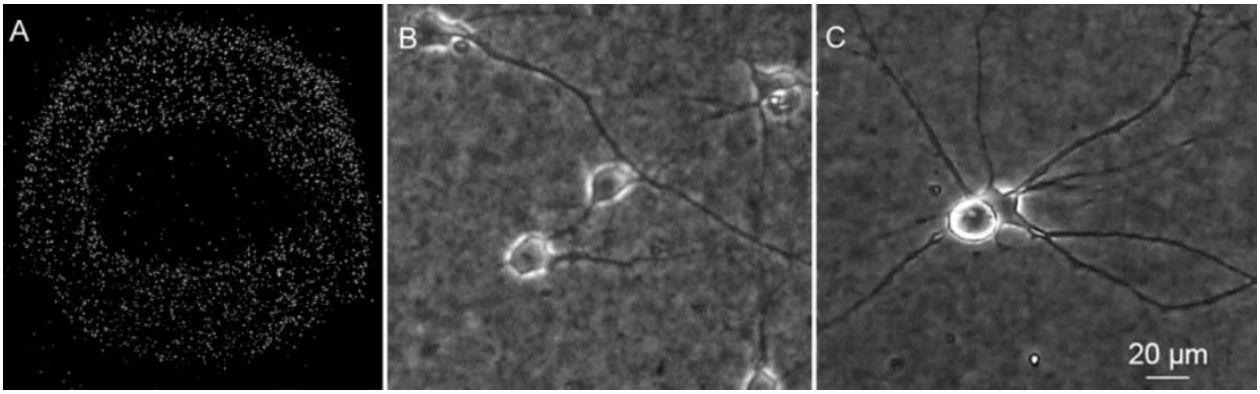


Figure 1.1 Primary cortical neurons were delivered by inkjet printing in the pre-designed ring patterns as indicated at day 1 utilizing bright field miscopy (a). At day 3 many of the neurons exhibited a characteristic neuronal morphology as seen with bright field miscopy (b). At day 7 the processes of two neurons have grown to about 40 μm in length (c). Magnifications: 40× (a), 100× (b, c) **[**1]

The complex, dynamic systems that make up human organs and tissues constantly change in response to their environment and outside stimuli. Because of their intricate compositions and complicated topologies, they are difficult to duplicate with conventional 3D printing methods. Additionally, the fact that most 3D bioprinting methods require a stable and level surface to deposit the bio-ink limits the range of applications and possible geometries (Fig. 1.2). This imposes a severe restriction on printing on curved or dynamic surfaces, such as living organs or open wounds. The ability to print on non-planar and dynamic surfaces would enable new applications for 3D bioprinting, such as repairing damaged organs or tissues directly on the patient's body, creating

customised implants or devices, or enhancing the functionality and aesthetics of already-existing structures.

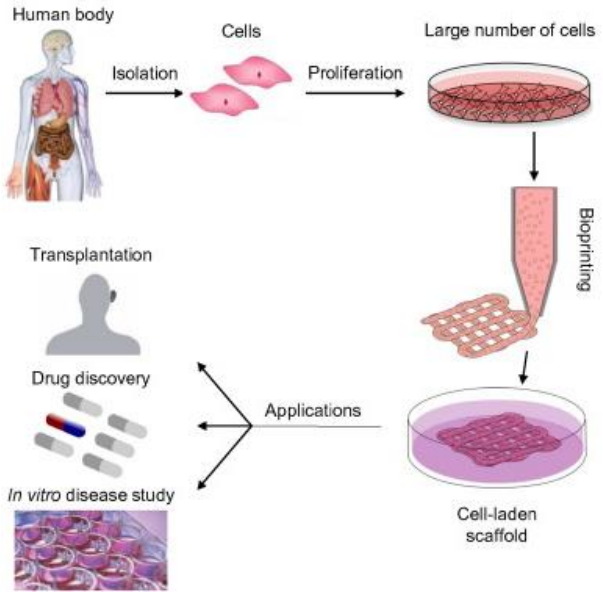


Figure 1.2 Bioprinting process and applications. For human therapeutic applications, the typical workflow of bioprinting would involve the isolation and expansion of human cells prior to printing the desired cell-laden scaffold. These scaffolds could then ultimately be used as therapeutic devices themselves, as a testing platform for drug screening and discovery, or as an in vitro model system for disease [2]

For instance, 3D bioprinting might provide skin grafts for burn sufferers, cartilage implants for those with osteoarthritis, or cosmetic improvements for people undergoing plastic surgery. However, in order to make use of these potential advantages, a trustworthy tracking system is required to guarantee the exact alignment of the bio-ink nozzle with the target surface throughout the printing process. A schematic representation of the difficulties and possibilities of 3D bioprinting on moving surfaces is shown in Figure 1.3. The quality and authenticity of the printed tissue may be impacted by the complicated curvature, deformation, motion, or roughness of the target surface. Additionally, the form and stability of the printed structure may be affected by the bio-inks changing viscosity, rheology, or curing qualities.

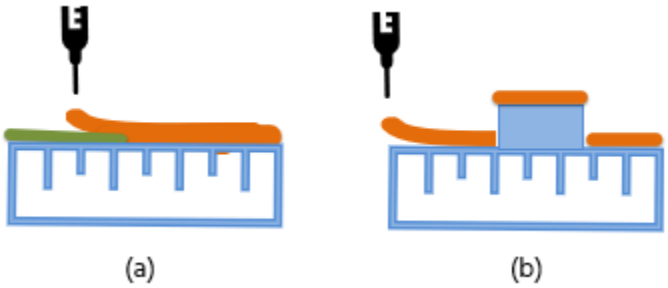


Figure 1.3 Bioprinting on non-stable surfaces: overlapping (a), and gap problems (b)

To be usable in the future, 3D bioprinting requires a tracking system that can recognise and adapt to these elements. Unsatisfactorily addressed in the literature is the unique and challenging research topic of creating a tracking system for 3D bioprinting on moving surfaces. The majority of tracking technologies now in use are intended for static or hard surfaces like bones or metal plates [3]. These systems use magnetic fields, optical sensors, or fiducial markers to find and position the bio-ink nozzle with respect to the target surface. However, these techniques are inconvenient for soft or dynamic surfaces like skin or organs because they may have asymmetrical forms, deformations, movements, or textures that might obstruct the markers or interfere with the tracking signals.

1.2 Research Motivation

Our work attempts to fulfil the need in unstable areas by pushing the limits of 3D printing. The two application areas of emergency usage and chronic wound treatment serve as the main driving forces behind this study.

Being able to respond quickly and effectively to medical emergencies in combat zones or emergencies might be the difference between life and death. Because of these unforeseen circumstances, current ways of providing medical supplies and managing on-the-spot wound treatment frequently fall short. The idea of 3D printing on moving surfaces is a trailblazing fix. This technique has the potential to significantly improve survival rates by enabling the in-situ bioprinting of wound dressings directly onto injured persons. This capacity is especially important

since it enables quick medical actions customised to the circumstances' exact requirements when stability and traditional medical settings are lacking.

Because they don't heal well with standard care, chronic wounds like diabetic ulcers pose a significant challenge to the medical community. Individuals with these illnesses experience chronic discomfort and are more likely to experience severe side effects. Our technology promotes better healing results, which guarantees ideal fit and adherence. Patients may heal more quickly and have a higher quality of life if the printed material can dynamically adjust to their motions, increasing comfort and efficacy.

The application of this technology in emergency and medical situations may potentially be profitable. It can cut down on waste and logistical expenses by enabling on-demand manufacture and minimising the need for large quantities of medical supplies. Reducing the frequency of hospital visits and long-term care costs might help manage chronic wounds and lessen the financial burden on patients and healthcare systems.

Investigating 3D printing on moving surfaces has the potential to progress additive manufacturing as well. It strains the boundaries of existing technology, necessitating the creation of novel materials, printing methods, and motion-tracking algorithms. These developments may have far-reaching effects, creating new opportunities for study and applications in a range of sectors.

1.3 Research Objectives

This thesis is dedicated to addressing the question of how to adjust a robotic hand to match the body movements and print tissues without collisions and gaps. The research objectives are:

a. Wound Area Detection: Identify and segment damaged tissue, isolate the wound area from background noise, and reduce image size for quicker processing.
b. 6-DOF Motion Detection and Estimation: Detect, track, and estimate body-surface movements: random and natural, significant and slight.
c. Printing Trajectory Generation: Generate an optimal printing trajectory layer-by-layer and provide customisability based on the material properties.

d. 6-DOF Robotic Arm Control: Develop an algorithm for a robot arm adjustment control system that integrates real-time feedback and safety protocols for emergencies.

1.4 The Layout of the Thesis

The thesis contains seven chapters and the references list:

1. **Introduction:** In this chapter, we review the history of 3D bioprinting, and discuss its promising potential for this field is for the future of the medical field, as well as current limitations, research motivation, and defined research objectives.
2. **Literature Review:** In the second chapter, we explore five distinct topics, each directly relevant to our research. We start with the assessment of additive manufacturing methods in the biomedical field, current results in multimaterial printing, researched approaches for in-situ printing, and evaluated different printing trajectory patterns in the first sub-section. In the second sub-section, we discuss the importance of the system setup, including lights and camera placement, and find the best fit of scanning methods for our research application. The third sub-section demonstrates the ways to segment the area of interest – wounds. The last sub-section of the literature review chapter reviews multiple ways to track objects, from optical flow methods to deep neural networks.
3. **Wound Area Detection:** Based on the first objective chapter, here we present and compare two methodologies for wound segmentation. The introduction section starts with the algorithm overview: input format expectations, discusses the multithreading approach and formation of the output. The first methodology reviews the pure Computer Vision approach. On the other hand, the second methodology reviews a Deep Learning model. In the end, we compare two methods based on the required processing time and final accuracy.
4. **6-DOF Motion Detection and Estimation:** The second objective is dedicated to discussing and elucidating the triangulation process to estimate movements and rotations in 3D. We split the process of estimations into three sections: feature extraction, tracking, and rotation estimation. In the following section, we validate our results with three different experiments: 1-DOF real-world movements, 6-DOF simulations, and real-world 3D rotation estimations.

5. **Printing Trajectory Generation:** The third objective, this section outlines the usage of multi-dimensional checkpoints as part of the printing path. In the second section, we reveal our algorithm for printing trajectory initialisation based on a simulated 3D mesh. The following section provides an analysis of the optimal path selection. Finally, we discuss how we update trajectories depending on the estimated 3D movements.
6. **6-DOF Robotic Arm Control:** In the chapter of the last objective, we bring an algorithm to operate a robotic arm with the consideration of dynamic commands and safety.
7. **Conclusions and Future Work:** The final chapter summarises the fundamental problems and findings in the formulated objectives, then presents an overall contribution to the field, considering all the performed experiments and results. Outlines the future roadmap.

2 Literature Review

2.1 Additive Manufacturing in the Biomedical Field

With the goal of improving cell viability and functioning, additive manufacturing (AM) technology is used in bioprinting to create complex structures that model natural tissues. Three main advantages of 3D bioprinting are the creation of functioning organs, tissue regeneration, and personalisation. For the replication of human organs to be implemented successfully, however, issues including scalability, resolution, printing area restrictions, cell survival, in vivo integration, and material compatibility must be resolved [4].

In the context of AM, resolution describes the printed object's degree of accuracy and detail. Smoother surfaces and better details come with higher resolution, yet printing at higher resolutions usually takes longer. To attain the required degree of detail without sacrificing efficiency, it is imperative to strike a balance between resolution and printing time [4]. Conversely, scalability describes the capacity to expand the manufacturing process or raise output volume. Scalability refers to the possible mass production of bioprinted tissues and organs. Researchers and manufacturers can improve the viability and efficacy of AM in making bioprinted tissues and organs for a range of medicinal applications by striking the correct balance between scalability and resolution [5]. This equilibrium is necessary for fulfilling mass production requirements in biomedical applications, where great accuracy and efficiency are required in the creation of complex structures.

Based on the above, it can be determined that printing speed is another aspect of bioprinting efficiency. Delays in the formation of complicated tissue architectures and manufacturing output, especially on unstable surfaces, can result from slow printing rates. One of the options to increase printing speed is using multi-nozzle systems for simultaneous deposition, adjusting settings, and enhancing material qualities for faster curing. Another option reviewed by researchers is a multi-material single-nozzle system for the same results as a multi-nozzle system [3]. Increased productivity, reduced potential for cell injury, and improved tissue quality result from faster printing, advancing tissue synthesis in regenerative medicine.

2.1.1 Multimaterial Printing

The review of multimaterial bioprinting approaches by Dikyol et al. (2021) [6] mentioned several significant milestones in the field of multimaterial bioprinting which have been achieved. These achievements: the reconstruction of vascular networks, integration of multiple cell types, development of advanced bioprinting platforms, and the creation of multicellular and multicomponent bioinks [6], demonstrate the potential of multimaterial technology for creating tissue constructs with biological complexity and functional properties, bringing 3D bioprinting closer to clinical applications.

Multimaterial printing is the technology of printing multiple materials in a single process. A variety of designs of nozzle systems are utilised in multimaterial printing to enable the deposition of multiple materials at once. Each system has its advantages and disadvantages (Table 2.1).

	Single nozzle	Stationary nozzles	Lifting nozzles	Multi toolhead
Contamination	Yes	No	No	No
Tool Interference	No	Yes	No	No
Added Inertia	No	Yes	Yes	No
Alignment Challenges	No	Yes	Yes	Yes

Table 2.1 Overview of material switching systems [7]

Based on the above, single- and multi-nozzle systems will be reviewed. The single-nozzle printhead improves the biofabrication process in tissue engineering in several ways. Such a design (Fig. 2.1) facilitates continuous printing of multiple materials, eliminating the need for switching between nozzles, which leads to reduced printing time. Additionally, it allows for precise control over the extruder positioning, resulting in accurate movements.

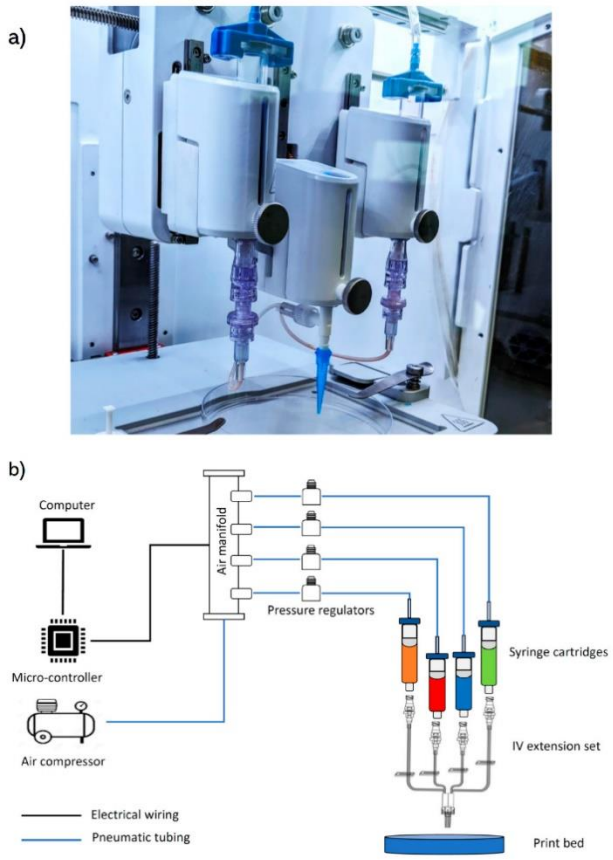


Figure 2.1 Assembly of the single-nozzle multi-material (SNMM) printhead on a 3D bioprinter (a), schematic diagram of the SNMM printing set-up (b) [8]

Park et al. (2015) [8] designed the multi-nozzle bioprinting system to simultaneously dispense thermoplastic biomaterial and hydrogel. The system consists of three axes of the x-y-z motion control stage and a 1-axis actuator for changing the nozzle (Fig. 2.2).

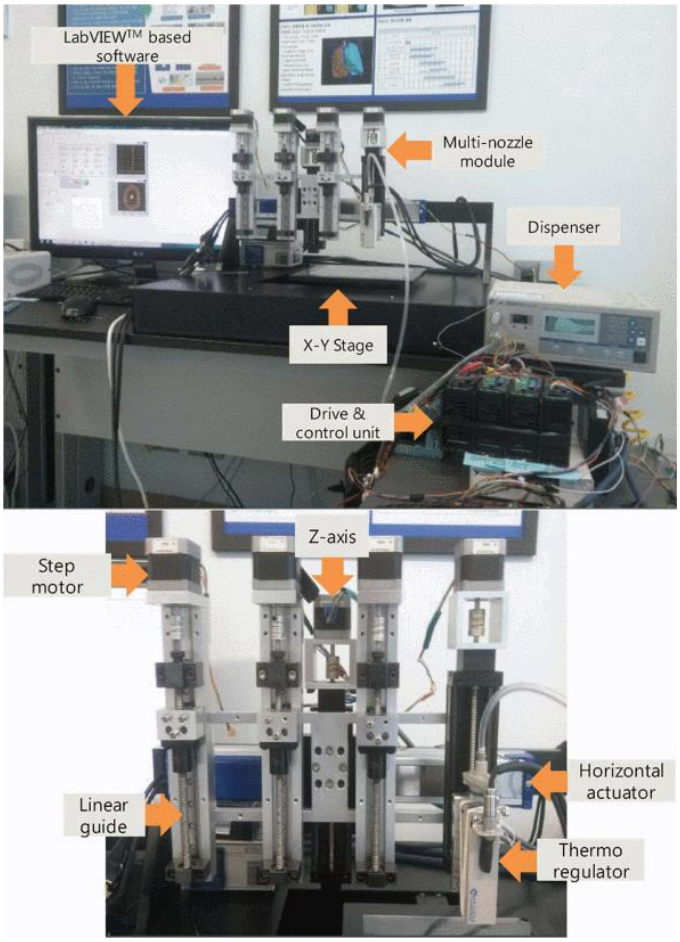


Figure 2.2 A multi-nozzle bioprinting system setting [8]

The control mechanism of the multi-nozzle extrusion system is efficient in printing processes through automatic filament changing features and multiple distinct nozzles. The system is designed to have multiple different filaments installed simultaneously. The changing of materials is done automatically, reducing the time needed to switch between them [9].

2.1.2 In-situ Printing

In-situ bioprinting proved its value for wound healing treatments [10], [11]. With the use of this technology, skin cells are directly delivered to the wound site, hastening the healing process and re-epithelialization. The material is delivered straight to particular wound regions, depending on size and topology, and replicates the layered skin structure. When compared to wounds treated

with conventional cell spraying procedures, bioprinted wounds show less contraction, which might result in superior esthetic outcomes [10].

Xie et al. (2022) [11] selected a robotic arm system as the in-situ bioprinting tool for depositing bioink onto a patient's wounds (Fig. 2.3). 3D models of the injury are reconstructed using computer-aided design software, and the printing routine program is generated with slicing software and loaded into the Robot Operating System (ROS) of the robotic arm [11]. The bioink is then in-situ deposited into the cranial defects of the rat "patients". The bioink is specifically designed for in-situ applications, addressing controlling rheological properties, adapting to the wound environment, and ensuring crosslinking and mechanical properties [11].

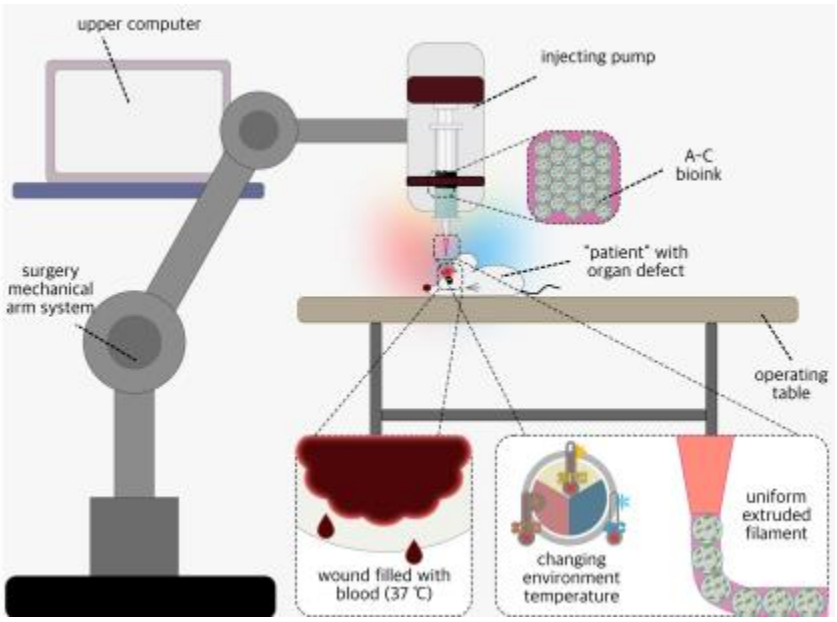


Figure 2.3 The process of in-situ bioprinting [11]

Another use of the in-situ method is the creation of custom-designed devices and structures that conform to the specific geometry and requirements of the biological surface, such as the skin, tissues, or organs. Zhu et al. (2018) [12] proposed an approach that combines direct ink writing with closed-loop feedback and computer-vision-based control techniques to create seamless interfaces for autonomously manufacturing multifunctional devices on moving freeform surfaces. This technique enables a wide range of applications in healthcare, biotechnology, and bioengineering.

2.1.3 Printing Trajectory Generation

Since a wound is an irregularly shaped surface, printed layers are not flat, either. The paper of Shembekar et al. (2019) [13] presents trajectory planning algorithms for 3D printing using a 6 degrees-of-freedom (DOF) robot arm that directly deposits material onto nonplanar surfaces. The main concept behind generating a robot path is to create a sequence of positions and tool inclination angles for the tool centre point (TCP) of the robotic arm [13]. The algorithm should consider the complex geometry of a wound and the capabilities of the 6-DOF industrial robot arm:

1) Robot waypoints and hatching pattern: the tool path is generated by dividing the surface into small patches and then developing a tool path for each patch. The hatching pattern (zig-zag, raster, spiral, contour), preselected by the user, is generated for each patch [13].
2) Tool orientation calculation: this step calculates the TCP orientation for each point on the tool path [13].
3) Trajectory parameters control: printing speed, flow rate, etc., are also selected by the user [13].

To reduce the transition travel distance, the travelling salesman problem (TSP) [14] might be resolved - the algorithm treats each printhead position as a "node" in the graph system. The problem is calculating the shortest possible route that the nozzle should take to cover all these points. Per each layer of the printing objects, the algorithm generates sub-paths and then combines them into a single global path using a greedy algorithm [14]. The greedy algorithm helps to reduce the path length by always moving to the nearest next point, thereby minimising robot joint movements. However, in complex geometries, a basic iterated greedy algorithm is vulnerable to early convergence and frequently gets stuck in local optimality [15]. Ying et al. (2022) [16] developed the adjusted iterated greedy to increase the exploitation power of the solution algorithm (the problem under horizontal and vertical ground motion), the initialisation and construction methods were modified, and a local search, based on the Tabu idea, was incorporated into the search operation (Fig. 2.4) [16].

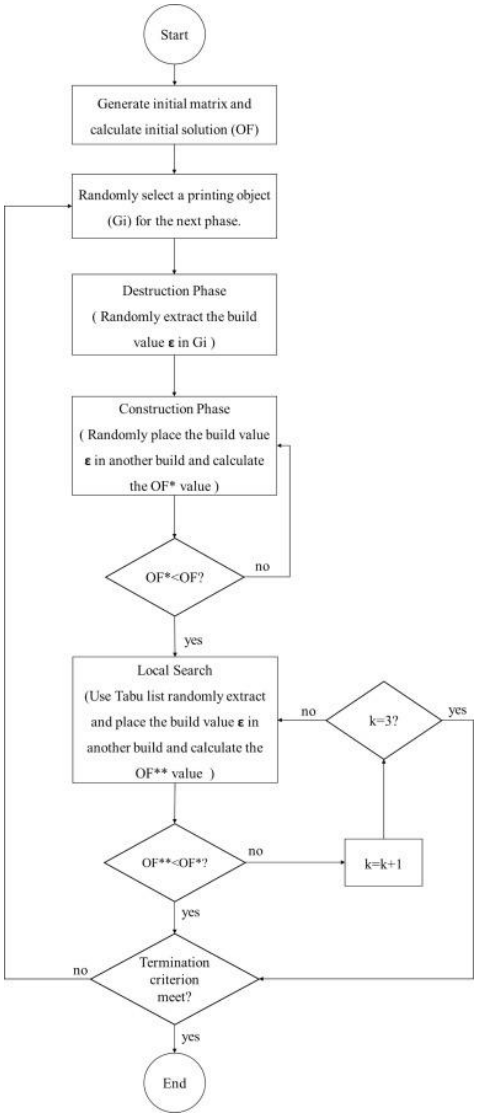


Figure 2.4 Adjusted iterated greedy algorithm **[16]**

Ganganath et al. (2016) [17] looked at this problem as a time optimisation problem. When traditional TSP algorithms focus on minimising the total travel distance, authors determined that, for 3D printing, the goal is to minimise the total traversal time. The cost of each edge in the modified TSP is defined in terms of time durations, not distances [17]. In the context of 3D printing, the trajectory includes both print segments (where the nozzle moves at a constant speed to deposit material) and transition path segments (where the nozzle moves as quickly as possible to minimise transition time). This requires the algorithms to optimise for these two types of movements

differently. In their work, they propose to evaluate three algorithms, random selection, Nearest Neighbor, and Christofides algorithm, individually to select the most effective per task.

Between different hatching patterns, the zig-zag trajectory is the simplest to design and operate [13], [14]. On the other hand, circular trajectories were found to be challenging in relation to complexity, error compensation, and maintenance (regular calibration required) [18].

2.2 3D Model Reconstruction System Setup

This section is partially published in: Zakharova, K.; Czekanski, A. 3D Bioprinting On Moving Surfaces Using Computer Vision. Canadian Society for Mechanical Engineering International Congress, vol 6, Sep. 2023.

Our tracking system is a complex product that has specific requirements. Besides the 3D bioprinter system and software, it is necessary to consider proper camera positioning, light setup choice, and 3D scanning approaches.

2.2.1 Cameras System

A multi-camera photography (MCP) system combines two or more synchronised cameras to capture pictures of a scene or object from various angles (Fig. 2.5). An MCP system is used to record a set of images to reconstruct a 3D model, providing comprehensive details on the form, texture, and structure of the subject. To guarantee thorough coverage from every viewpoint, the placement of cameras and the overlap between their fields of vision may be adjusted.

Camera number	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
α (degree)	0	12	24	36	48	60	72	84	96	108

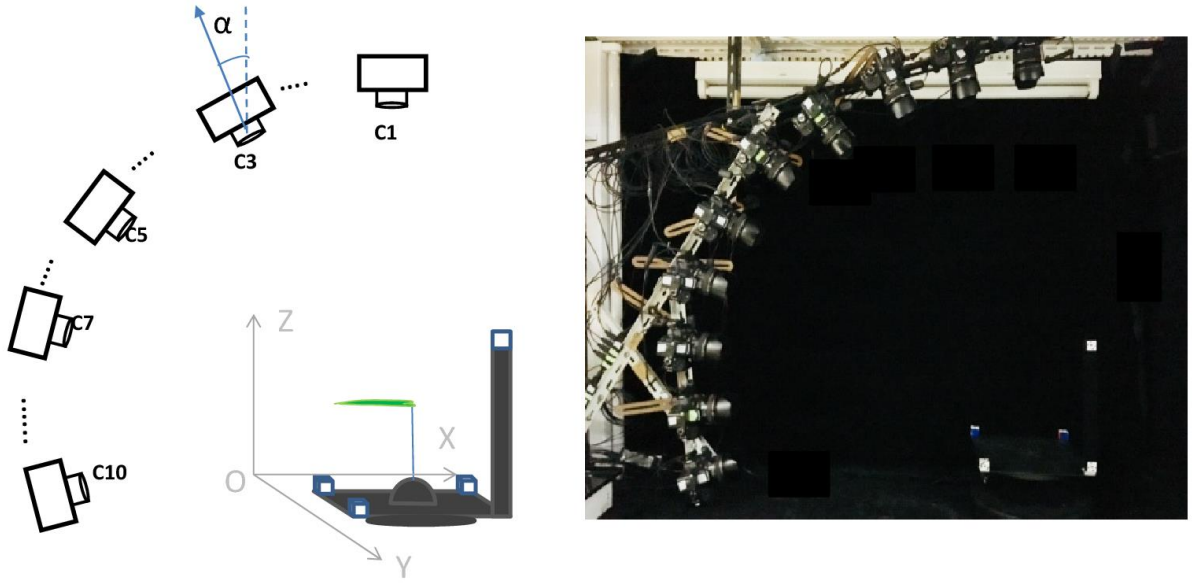


Figure 2.5 MCP system (right: the actual system; left: its schematic; C1–C10 represent 10 cameras, respectively; α is the viewing zenith angle) **[19]**

Using an MCP system with 10 digital single-lens reflex (DSLR) cameras and a rotary table for the case of plant reconstruction yielded compelling results in 3D segmentation described in the work of Lu et al. (2020) [19]. The 3D mesh model of a single leaf, reconstructed using images taken at viewing zenith angles (VZAs) from 12° to 60°, had fewer undetected or unstable regions, demonstrating that selecting appropriate camera angles enhances leaf reconstruction accuracy. When merging 3D dense point cloud models from images taken at each optimal VZA, the 3D mesh model of the whole plant showed low error percentages (Equation 2.1) for leaf area, length, width, and stem height and width. The study compared two reconstruction methods: Method 1, which involved building 3D dense point cloud models from images taken at optimal angles and then merging them, proved more accurate and efficient than Method 2, which built the 3D mesh model from all camera angles [19]. In their setup, the camera at 36° showed the smallest area error percentage (1.02%), meaning that it provided the most detailed coverage of both the side and top parts of the leaf [19].

$$Leaf\ area\ error\ percentage = \frac{|Estimated\ area - True\ area|}{True\ area} \times 100\% \qquad (2.1)$$

2.2.2 Lights Requirement

The light setup is another requirement for the system. It is necessary to keep the information from the video frames by reducing the harshness of the shadows and wound reflections. The ways of this are subject to study.

1) Multiple light sources: using multiple light sources with different angles and intensities can help illuminate the area of interest evenly and remove all visible shadows, and the constancy of the brightness is also reliably guaranteed [20].
2) Diffused light: using diffused light, such as through a softbox or translucent panel, can help to reduce harsh shadows and create more even lighting [21].
3) Reflectors: placing reflectors around the surgical table will also help bounce light, brightening the shaded areas [22].
4) Polarised filter: while a polarised filter does not affect the light, it definitely works to fight reflections in output images [23], [24]. The polarized lens blocks certain orientations of light waves.

It is possible to remove or reduce the appearance of shadows in images using computer vision (CV) techniques [25], [26]. Still, it is not a trivial task, and it can be challenging to remove all shadows while keeping all the information completely. The main approach to eliminating or reducing shadows is by image processing techniques: image enhancement, restoration, and segmentation. However, the main issues can arise when attempting to remove shadows in this way. All of them are very important to our project:

- Information loss: shadows can contain important information about the shape and position of objects in an image. Obliterating shadows can lead to losing important information and affect downstream tasks' performance.
- High computational cost: the process of removing the shadows from an image can be computationally expensive and time-consuming.

2.2.3 3D Scanning

Any harmed area being working on is a 3D object. It is an important first step to getting high-resolution 3D mesh for further system processing.

Generally, there are two ways to find out the 3D shape of an object: scanning (LiDAR, laser, structured light, photogrammetry) and using Deep Learning (DL).

1) LiDAR and laser scanner: the LiDAR system emits laser pulses toward the objects; when these pulses bounce back to the sensor after hitting an object, the system measures the time it takes for each pulse to return – the distance between the sensor and the object. These techniques are accurate and reliable but not a fit for our case. One of the main reasons is the resolution. Laser scanning typically captures a low density of points (for example, 5 points per square meter) [27], [28] while small to mid-size shapes are the focus.
2) Structured light: this technique projects patterns of light onto an object and uses a camera to record how the pattern is deformed by the surface. The deformation is used to calculate the 3D shape of the object. It's used to scan objects with complex shapes or surfaces that are difficult to capture with other techniques [29], [30]. There are various types of scanners, from huge setups to handheld apparatus [31].
3) Photogrammetry: it uses mathematical algorithms to extract 3D information from 2D images and calculates the spatial relationships between objects and their surroundings. It involves the use of two or more cameras positioned on a frame around the target area [32]. The resulting measurements from images are found to be highly correlated with the traditional estimates [33]. However, it was mentioned that more extensive wounds can cause issues because it might be challenging to photograph the wound from a single viewpoint [32], [34]. Additionally, moisture or partially hidden wounds are difficult to capture due to reflections and shadows or blocking subjects [32], [35].

There are neural network algorithms that can build 3D models from 2D videos or images [36], but most of them are computationally and time expensive or trained on large/convex objects, which is the opposite of our problem. However, another purpose for using DL was found – to implement scan post-process. Any of the scanners capture the whole picture of the surface - not only the wound but the background, too. As the alternativity to the DL model, there are different foreground extraction algorithms, such as GrabCut [37]. This is a fast solution choice, but it might

be inaccurate (cropping the suitable pixels, keeping background) or require additional human efforts.

2.3 Wound Segmentation

This section is partially published in: Zakharova, K.; Czekanski, A. Wound Segmentation With Semi-Supervised U-Net Model For Unseen Perspective Tracking. Canadian Society for Mechanical Engineering International Congress.

Accurate wound segmentation in medical imaging facilitates accurate diagnosis severity and effective therapy monitoring [38], [39], [40], [41], [42]. DL models demonstrate excellent results for wound segmentation [38], [40], [41]. There are three ways to use DL for wound assessment:

- Wound detection: localise wound on the whole frame and crop it for further processing: classification, extracting harmed area characteristics [39], [41], [43]. YOLO models [44] are the best option in terms of time processing and accuracy [39], [40], [41].
- Segmentation: segmentation is a pixel-wise masking, specifically in the wound case. It delineates the wound boundary, separating the wounded area from the surrounding healthy tissue or background [38], [40], [43], [45]. By having the mask of the wound, its total area can be estimated [38], [40], its shape (round, ragged edges) can be recognised, or they can be categorised: wound type (bruises, burns, infected, etc.) [43], severity (surgery is/not required) [45]. The most popular and one of the most productive Convolutional Neural Network (CNN) architecture for wound segmentation is U-Net [38], [40], [42], [43].
- Detection + segmentation: this approach combines both strategies. First, the detection model localises the wound bed region within the image and, after, is used as a precursor to the segmentation step. The detection step helps identify the area of interest for subsequent segmentation. Scebba et al. (2022) [38] mention that such a method improves accuracy by better generalisation to out-of-distribution data and can handle reduced training data without compromising performance. By first detecting the wound region and then segmenting it, the models can adapt to different imaging conditions, making the approach versatile and applicable across diverse datasets [38], [40].

One of the main problems of wound segmentation is obtaining datasets, especially with labels, for training these models (Table 2.2).

Dataset	**Data Description**		
	Number of images	**Number of types**	**% of labels**
Deepskin [42]	1564	1	10
DFU dataset [46]	1459	1	100
AZH dataset [43]	830	4	0
AZHMT dataset [43]	1088	4	100
Boissin et al. [45]	1105	1	100
WoundsDB [47]	188	1	100

Table 2.2 Datasets with wound images and masks: number of images available, present wound types, and percentage of labelled data to the whole dataset

Additionally, in Table 2.2, open-source medical image libraries (such as Medetec [48]), which contain diverse photos of multiple wound types but no labels, can be added.

The majority of the dataset creators mention that all their data was labelled manually by the medical experts [38], [40], [42], [43], [46]. Semi-supervised learning is a solution for overcoming the scarcity of labelled data in medical imaging tasks [42], [49], [50]. Semi-supervised learning (SSL) is a category of Machine Learning [51] placed between supervised and unsupervised learning. SSL is used on conditions having a small amount of labelled data and a large amount of unlabelled data (Fig. 2.6).

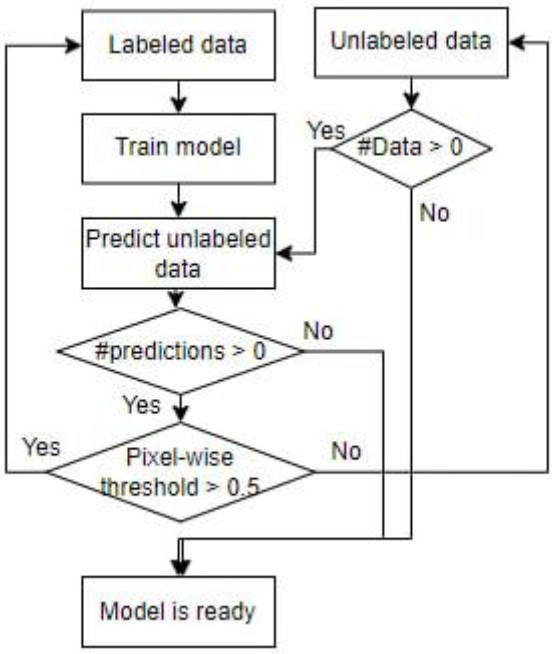


Figure 2.6 Iterative training process of SSL models [51]

2.4 Tracking Systems

Optical flow represents the pixel motion between sequential frames in a video stream as a dense space of displacement vectors [52], [53], [54]. It tackles 2D motion, capturing pixel movements within the image plane between video frames [54]. The fundamental assumption behind optical flow is that the intensity of a pixel remains constant as it moves from one frame to the next, namely brightness constancy [52], [53]. This assumption brings a challenge in illumination varying in the scene [52]. Besides the illumination changes problem, classic optical flow struggles with tracking high-speed movements and handling occlusions [52].

The future state of a movement can be predicted using the Kalman filter [55], [56]. KalmanFlow 2.0, by Bao et al. (2019) [57], utilises a context-aware Kalman filter - a filtering approach that dynamically adjusts its parameters based on contextual information extracted from the processed environment [57]. This algorithm quickly predicts and tracks object movements in the context of optical flow estimation tasks by presenting pixel motion flow as a second-order time-variant state vector and optimising the estimation based on measurement and system noise levels [57]. The Deep feature matching method combined with the Kalman filter and incorporation of an

optical flow tracker (DK-Flow by Chen et al. (2019) [56]) performs better compared to DK-tracking methods that assume constant velocity movement [55], [56]. Deep features are extracted by a CNN [58] and build matching relationships between candidate objects detected in the current frame and tracked targets [56]. Zheng et al. (2012) [59] extract features with scale-invariant feature transform (SIFT) and determine whether to update the object state based on the feature points' loss rate [59]. Template matching is used in their object tracking algorithm to compare the features extracted from the target area with a reference template to localise and track the object in each frame despite occlusions or position changes (scale or rotation) [59].

Karami et al. (2017) [60] provided a complete review of different feature matching methods, including performance comparison for distorted images:

- SURF (Speeded-Up Robust Features) [61]: SURF was designed to increase speed in feature detection and matching, however loses its performance (up to 15%) in rotation, noised fish eye distortion to its ancestor SIFT, while being at least twice faster [60].
- SIFT [62]: the slowest method from the list; at the same time, it is leading in accuracy, showing the top match rate for every distortion imaging, except scaling [60].
- ORB (Oriented FAST and Rotated BRIEF) [63]: offers fast feature extraction and matching combined with modified rotation invariant BRIEF (Binary Robust Independent Elementary Features [64]) [63]. ORB is significantly quicker than SIFT (two orders faster) and SURF (an order of magnitude faster), making it suitable for real-time applications on low-power devices such as smartphones [63]. ORB performs the best for different scaling images but is inferior in rotation and noise and depends on constant pixel intensity [60].

2.4.1 Transition from 2D to 3D

2.5D is a representation of a scene that captures both image information (2D) and partial depth or geometric information (3D) [52]. 2.5D focuses on understanding the relationship between objects in a scene with depth and occlusion and the camera's viewpoint as a reference point [65]. Scene flow extends the concept of optical flow into 3D, capturing motion information and the displacement of 3D points of objects in the scene [52].

In CV, the process of triangulating involves projecting a point's position in three dimensions onto two or more two-dimensional pictures that were obtained from various angles [66]. Epipolar geometry refers to the geometric correspondence between two viewpoints of the camera [67]. With the idea of epipolar lines, an object's location on one picture may be connected to its position on the other when it is viewed from two distinct perspectives. By using these lines, the search space for points that coincide between photos is reduced. Camera calibration is necessary to know the extrinsic (rotation and translation) and intrinsic (focal length, primary point, distortion coefficients) properties of the cameras [68]. Mapping 2D picture coordinates to 3D coordinates is made possible by calibration. Structure from Motion (SfM) uses triangulation to take a sequence of 2D photographs and reconstruct the 3D structure of the scene from those locations [69]. It is possible to estimate the 3D locations of these characteristics as well as the camera motion by tracking them throughout these photos. By analysing the difference between a point's position in the left and right camera views, triangulation is employed in stereo vision to determine a point's depth [70].

2.4.2 3D Tracking Multicamera Systems

Gorjup et al. (2021) [71] present a method for measuring high-frequency 3D vibrations of complex-shaped specimens using still-frame cameras and the Spectral Optical Flow Imaging (SOFI) method. They extend the SOFI method to 3D operating-deflection-shape (ODS) measurements, allowing full-field displacement measurement in three dimensions. By utilising harmonically controlled illumination and a single still-frame monochrome camera, the authors demonstrate that complex-shaped specimens can be measured without the need for multi-camera, high-speed imaging hardware [71].

2.4.3 Deep Learning Models

OmniMotion (2023) [72] and DINO-Tracker (2024) [73] are two state-of-the-art DL trackers that exist today. OmniMotion generates a 3D model of each item in a video scene and uses these models to track the movement of the objects over time. It builds this model using a representation called a "quasi-3D canonical volume" [72] (Fig. 2.7), which is like a 3D map showing where different parts of the objects are located in 3D space.

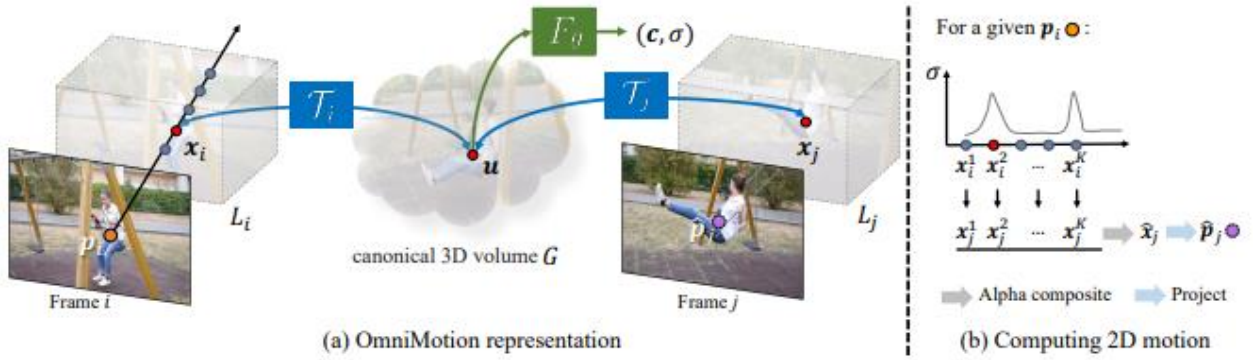


Figure 2.7 OmniMotion overview. (a) OmniMotion representation of a canonical 3D volume (G) and a set of bijections (Ti/Tj) that map between each frame's local volume (Li) and G. (b) To compute the corresponding 2D location for a given points, they shoot a ray into Li and sample a set of points, which are then mapped first to the canonical space to obtain their densities, and then to frame to compute their corresponding local 3D locations. These points are then alpha-composited and projected to obtain the 2D corresponding location [72]

DINO-Tracker uses a different approach. It tracks features by using a novel technique for deep spatial analysis and temporal coherence. DINO-Tracker uses complex attention mechanisms and contrastive learning algorithms to keep track of object features over time and space [73] (Fig. 2.8).

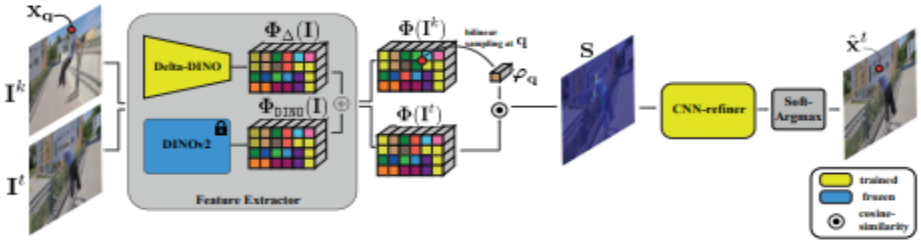


Figure 2.8 Feature extraction from a reference frame I^k, and a target frame I^t [73]

This technique helps the model to gather detailed information and even the slightest changes in the appearance of objects, even under severe occlusion, leading to currently the best tracking [73] and 3D space analysis. Although these models are accurate, they are not fast enough for real-time application, lacking the ability to analyse and respond instantly.

3 Wound Area Detection

Summary: This chapter discusses the identification and segmentation of wounds using two methodologies: traditional computer vision (CV) algorithms and deep learning (DL) models. It compares these approaches according to processing time and accuracy, focusing on the Dice coefficient metric. The setup and synchronisation of the cameras, the specifics of each methodology's implementation, and an assessment of their effectiveness are all covered in this chapter. It concludes that for real-time wound segmentation tasks, particularly U-Net architecture, performs better in terms of accuracy and efficiency than the pure CV method.

3.1 Introduction

Identifying the region of interest (ROI) on the patient's body, which is a wound in our case, that the camera system should focus on is the first objective's aim. In this chapter, we review two methodologies: one proposes to use pure CV algorithms, and another proposes to use a DL model. We compare these methodologies based on the accuracy (Dice coefficient metric) and processing time measurements. The objective is necessary for performance, and noise reduction is the following objective (movement detection).

ROI segmentation identifies a wound and crops it to decrease the size of the input frame to a bounding box of the wound. As a result, the input frames in further processing have the smallest shape. In CV, RGB (red-green-blue) images are represented as three stacked matrices of pixel values per colour (or image channel) with dimensions equal to the image resolution. A greyscale image of Full HD resolution would be represented as a one 1920x1080 matrix with values ranging from 0 (black) to 255 (white). The lower the dimension of the image, the faster the processing is (Fig. 3.1).

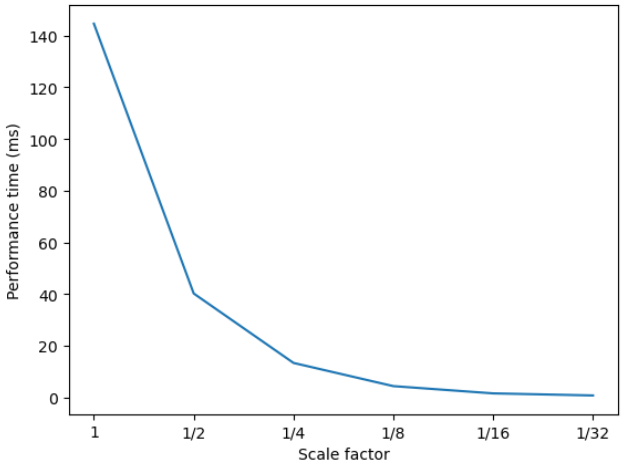


Figure 3.1 Performance time decreases exponentially as the scale factor of a Full HD image decreases

As was mentioned above, the wound segmentation separates the wound from the background noise, printing nozzle, and printed material in the ROI mask to ignore movements of such classes, which may interfere with the valid output results in the next objective.

Before going deeper into all the processes on the programming side of the objective, we should start with a description of the camera setup. For our setup, we used three webcams to capture wounds in 3D. The cameras are placed at the designated angles (Fig. 4.3): 0° (top), 36° (side), 84° (bottom) [19]. The top camera has an overarching view of features on top of the surface, the side one additionally captures part of the features, invisible to the top camera, on the side of the surface, and the bottom camera has a complete perspective on the side features but also a slight overview on the top features. With the focus on performance, three cameras are just enough to capture all the rotations, so more cameras would be an excess.

Before cameras capture their first frame, they need to be synchronised. Synchronisation is required to get Nth frames of the top/side/bottom cameras with the same timestamp *t*. We implemented the synchronised capturing using the Python multithreading library. This allows for the capture of frames and their processing from each camera independently with a separate process (thread) (Fig. 3.2).

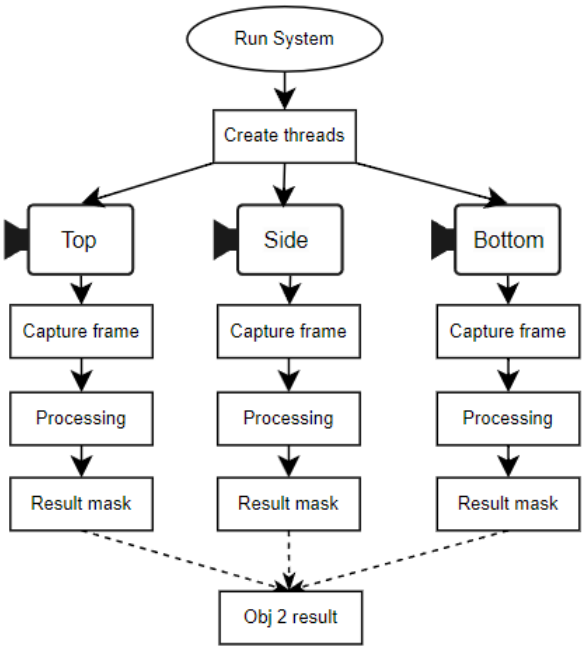


Figure 3.2 Diagram of multithreading processing. One thread per camera. As a result, the system sends 3 wound masks to the Objective 2 (Motion Estimation) processing block. The final block returns one result (displacement) by analysis of 3 individual masks

Now, when the system is prepared on the hardware and software sides, the main process begins with the acquisition of every 30^{th} frame per camera. Considering each camera has 29.91 FPS (frames per second), we capture and process every second of the live stream. We capture frames as a grayscale 1-channel image, as all other processing functions do not accept RGB images.

The following image processing and core functionality (harmed area segmentation) are described in each methodology section separately. After the processing, we crop a rectangle area around the segmentation mask.

A cropped ROI with a wound inside it is an output of the current objective (Fig. 3.3).

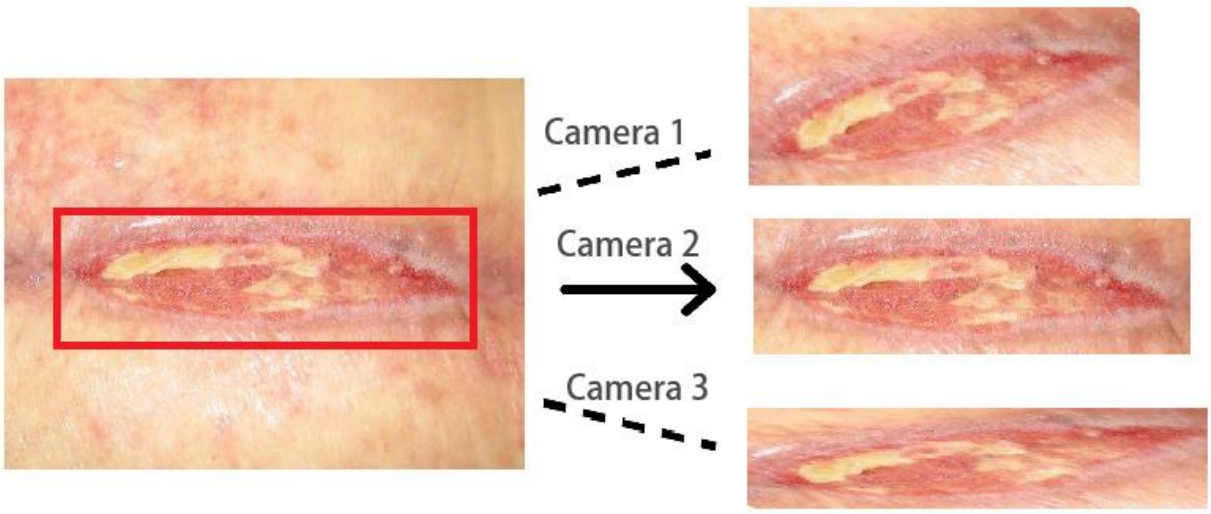


Figure 3.3 Cropped ROI from each point of view

But before stepping into the next process, the result must be prepared for the second objective to be able to estimate ROI displacement. With just one cropped frame, we can't estimate any movements, as the target object is always in the centre. To deal with this, the system remembers and passes further two cropped ROI images: the latest ROI and the previous one. The coordinates, or bounding boxes (Fig. 3.4), of those ROI should overlap and have the same dimensions. We leave an option to extend the boundary for the desired number of pixels.

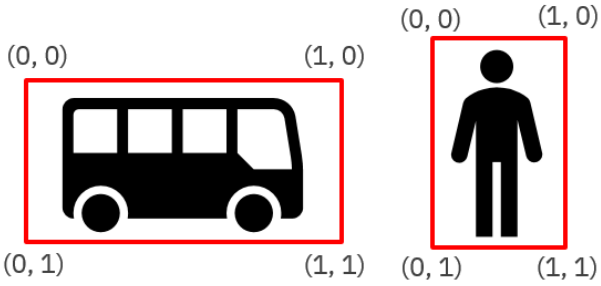


Figure 3.4 Bounding boxes coordinate description with the horizontal and vertical orientation examples

The united boxes keep the coordinates as follows: the top-left index equals the top leftmost index of the current or previous boxes, and the same goes for the bottom-right index – to the bottom rightmost. So, in the end, we have two small windows instead of the full frames (Fig. 3.5).

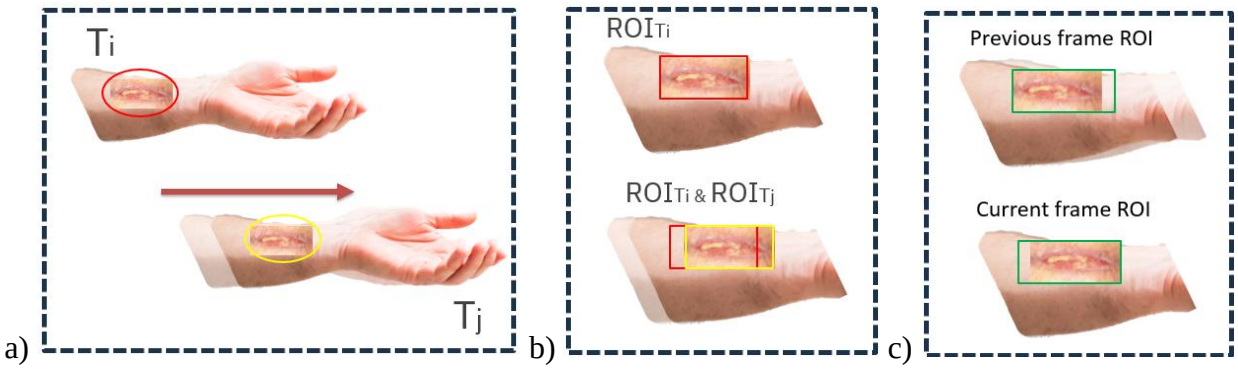


Figure 3.5 Cropping ROI a) captured movement between frames Ti and Tj. Ti is a previous frame, Tj is a current frame, b) red bounding box detects the previous frame ROI and yellow – current one, in the bottom example, we can see how two boxes overlap, c) cropped ROI for both frames is the output of the current objective

Wounds have specific features which other objects may not have. Lesions usually have various shapes that are impossible to predict. Thus, it is difficult to simulate wounds close to the real harmed area. To demonstrate our results, we used real wound images from an open-sourced dataset – AZH dataset [43]. The dataset includes 830 images and masks of wounds. For the comparison metric, we used the Dice coefficient (DC). DC is a pixel-wise comparison between the predicted mask (pred) and ground truth label (true) (Equation 3.1). DC score ranges from 0 to 1, where 1 means perfect overlap between the predicted and ground truth masks and 0 means no overlap at all. A higher DC indicates better similarity between the predicted and ground truth binary masks.

$$DC = \frac{2 * |pred \cap true|}{|pred| + |true|} \tag{3.1}$$

To compare time processing, we run the *%timeit* function in Jupyter Notebook (interactive computing platform). Time measurements are done for several wound pictures and measure only the segmentation process, excluding other pre- and post-processes as they run, irrespective of the methodology chosen.

3.2 Shape Matching (Methodology 1)

Wounds can have unpredictable shapes. With CV techniques, we can overcome this limitation. By having a reference to the wound, we can run a comparison algorithm that will look for our pattern on the video stream.

In this methodology, we employed pure CV algorithms for shape matching to process wound contours. Behind this idea, we detect wounds on the frame by their contour. For this module, we directly use an output from a 3D scanner – 3D mesh. To get the 2D contour from a 3D figure, we convert the figure into a numerical array. Obtaining the contour relies on determining the top part of the mesh (the top layer). The outline of the top layer serves as the shape pattern, which we save for the next operations. After, we find the same shape in the image and segment it. The complete process is illustrated in Figure 3.6, detailing each step from input to result.

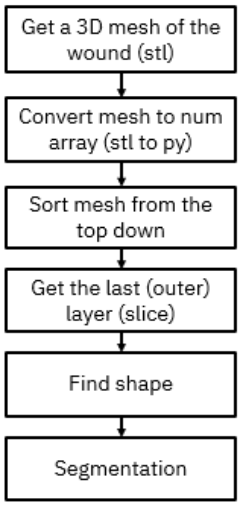


Figure 3.6 Diagram of the process for the shape matching module

To test our design, we simulated a 3D scan (Fig. 3.7). Our 3D simulation is a 1000-point cloud mesh. The mesh has an elliptical shape. The radiuses of the mesh vary from top to bottom. XY radiuses for the top layer are (0.25, 0.8), and the bottom one is (0.1, 0.15). The depth of the mesh is 0.3.

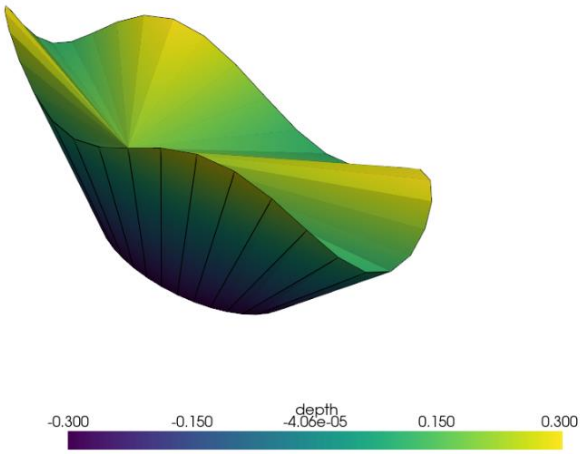


Figure 3.7 Example of input mesh from the 3D scanner. We simulated a 3D mesh of the wound with the PyVista library [74]

To transform the 3D mesh into a usable format for analysis, we converted it to a stack of 2D views by splitting vertices into horizontal layers. To split layers, we had a step N, representing depth level. With the highest level, we managed to remove basins and focus only on the highest points. Based on those, Figure 3.8 provides an approximate shape contour, which we use as a pattern.

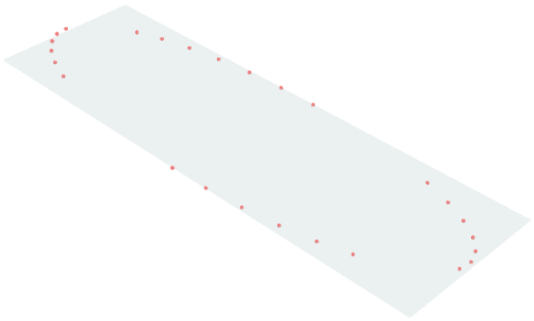


Figure 3.8 2D view of the top layer of the mesh

To validate our approach and to compare two methodologies, we used actual wound images and corresponding ground truth masks (Fig. 3.9). The masks consist of 0's and 1's, where 0 is the background, and 1 is the target area. All the images were labelled manually by wound care professionals.

Figure 3.9 Example of wound images (top row). Ground truth masks of the wounds (bottom row)

Below, we provide examples of the input patterns. Patterns appear as a solid or dashes line on a zero 2D array (Fig. 3.10).

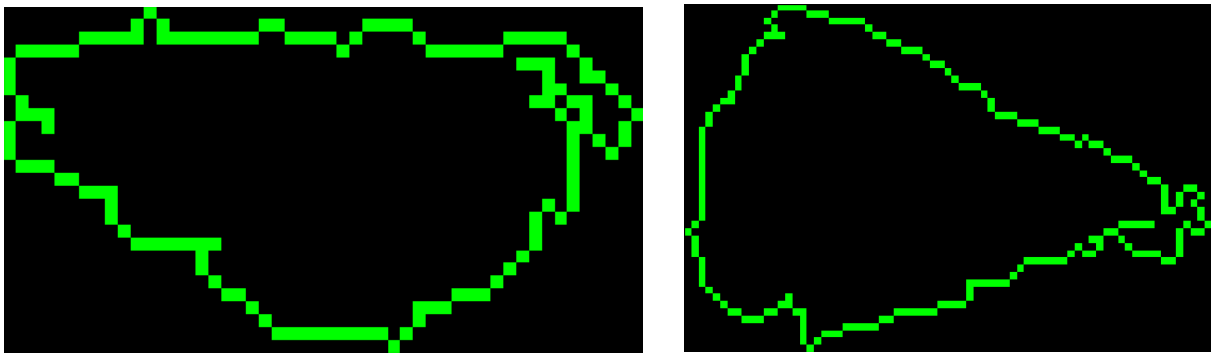


Figure 3.10 Input contour examples (patterns)

Following these examples, Figure 3.11 illustrates adaptive threshold masking. The adaptive threshold is a part of the OpenCV library [74]. This method is especially helpful in situations when illumination varies across different regions of the image. Adaptive thresholding modifies the threshold value according to the local pixel neighbourhood, in contrast to basic thresholding, which utilises a global threshold value for the whole picture. However, this method also detects a

significant amount of noise both around and inside the wound. We tuned the method with several parameters: a maximum intensity value of 255 was assigned to pixels exceeding the threshold; we selected the bigger block size of 11 to minimise details in the thresholded image; and the threshold value was determined using a Gaussian-weighted sum of the neighbourhood values, with a constant C set to 2.

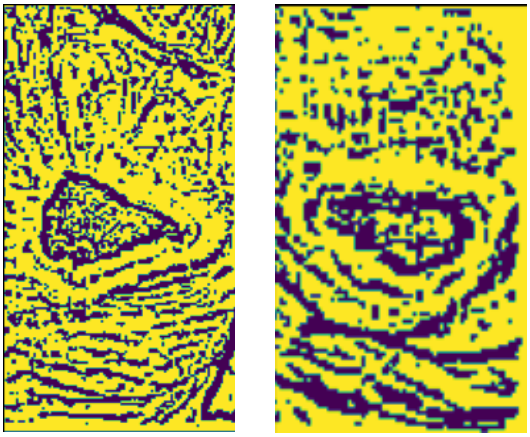


Figure 3.11 Adaptive threshold masks. We can notice the contour of the wounds, which matches patterns from Figure 3.10 accordingly

To challenge our methodology, we included more complex patterns and images in our experiments (Fig. 3.12).

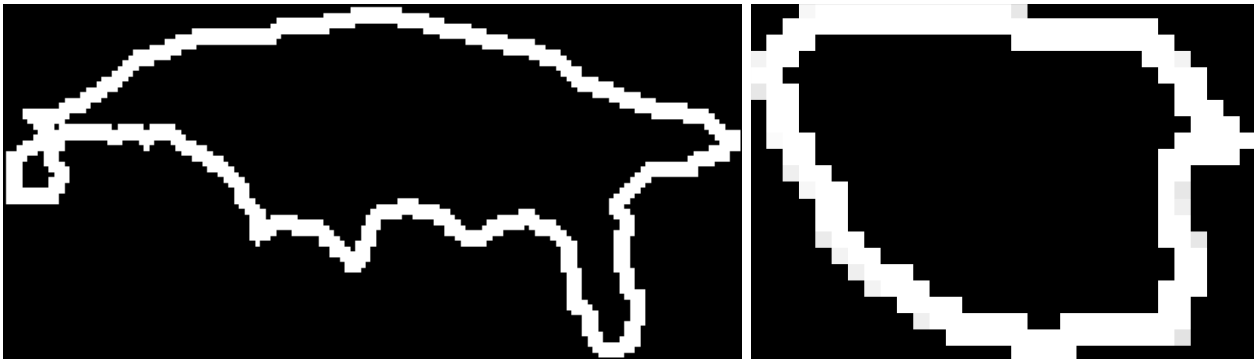


Figure 3.12 Examples of more complex patterns (Fig. 3.9 column 3,4)

Unfortunately, our method showed instability when handling noisy and uneven wound shapes, especially when wound edges were not distinctly separable from healthy skin by colour

(Fig. 3.13). This instability was particularly evident when the colour differentiation between the wound and healthy skin was minimal, leading to inaccurate contour detection.

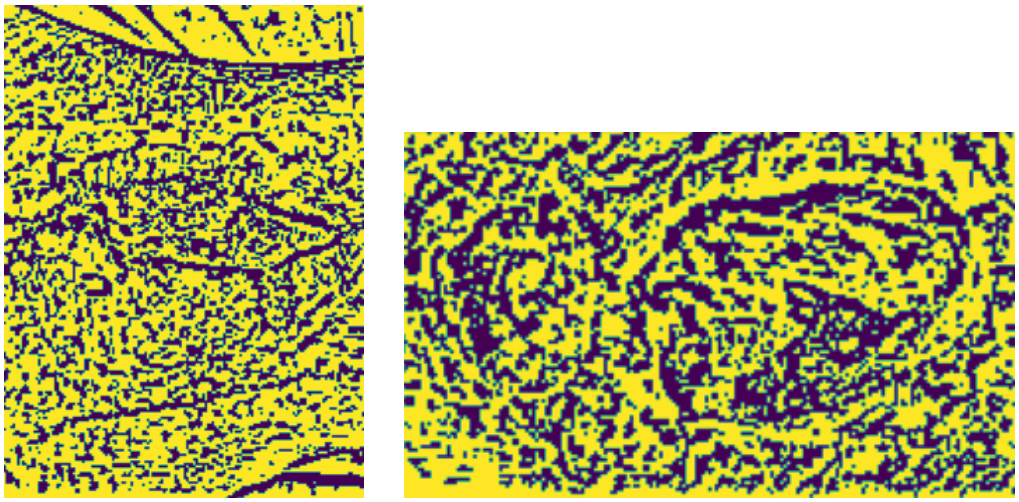


Figure 3.13 The method isn't stable with the noisy and uneven wound shapes when the edges of the wound aren't clearly separatable from the healthy skin by colour scheme

To address this challenge, future implementations of the methodology should incorporate additional image processing techniques designed to improve colour differentiation and edge detection. Although we tried different blurring methods to smooth the images and reduce noise, it also smoothed contours, making them difficult to distinguish. Similar issues arose when we attempted to enhance the contour edges, as these enhancements also distorted the natural shapes, complicating accurate matching.

Finding contours and matching shapes are the main operations that are performed in every video frame iteration. *findContours()* is an OpenCV method for detecting the contours in a binary image (threshold mask). Our choice of parameters in the function consisted of the RETR_TREE mode to find all the contours and CHAIN_APPROX_SIMPLE, which keeps a simplified version of the contour to save memory and speed up processing. *matchShapes()* compares two shapes, one shape is one of the obtained contours from the previous method, and the other one is a predefined pattern. *matchShapes()* returns a similarity score, which we use to select the most similar contour from the pool of the found ones.

3.2.1 Results and Discussion

The results of our shape matching methodology are summarised in several key figures. Detected contours on input images are shown in Figure 3.14. In this figure, the detected contours are overlaid on the original grayscale images, highlighting successful matches.

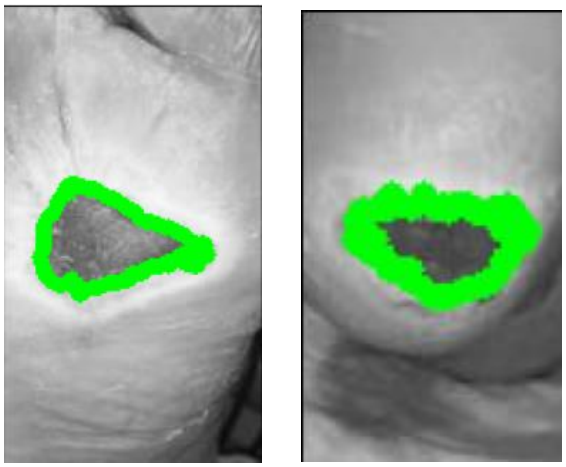


Figure 3.14 Detected contours on the input images

Unfortunately, some other attempts to match patterns with the threshold mask were unsuccessful, as illustrated in Figure 3.15. These unsuccessful attempts occurred due to the high noise levels in their threshold masks (Fig. 3.13).

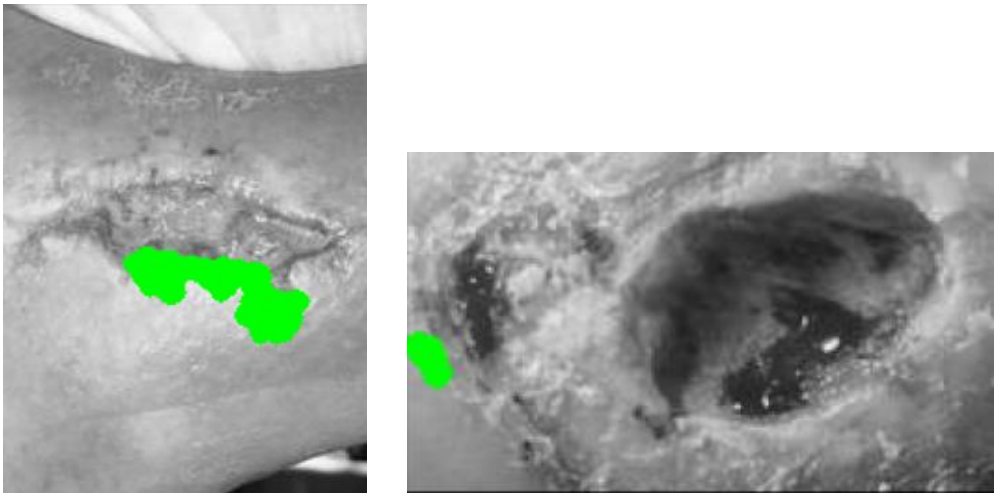


Figure 3.15 Example of a failed contour finding

Interestingly, the performance time of the shape matching algorithm did not depend on the amount of noise or the size of the pattern (Fig. 3.16).

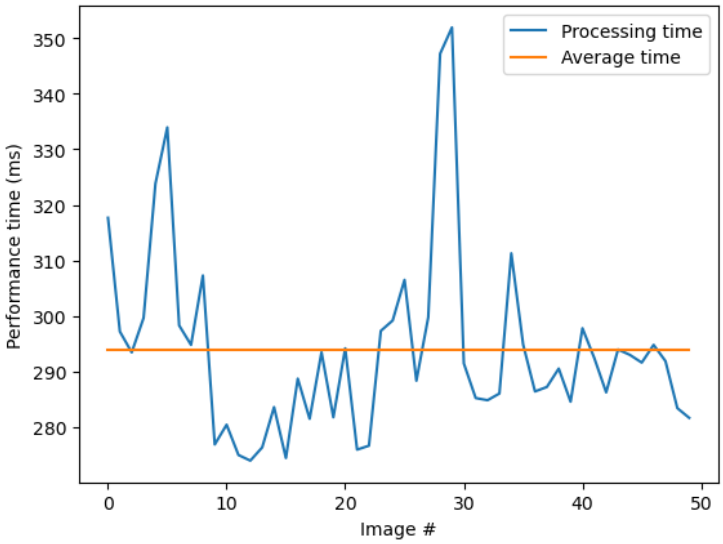


Figure 3.16 Performance time of shape matching algorithm on 50 wound images. Measured on CPU (AMD Ryzen 7 5825U)

The final accuracy of our method, assessed using the Dice coefficient metric, was 0.48. This score is an average of 50 examples, with individual scores of 0.93 for good examples (Fig. 3.14) and 0.12 for bad ones (Fig. 3.15). The high variance in these scores underlined the method's sensitivity to wound complexity.

3.3 Deep Learning Segmentation Model (Methodology 2)

Unlike the previous methodology, where we used only CV methods, in this section, we provide a comprehensive description of our approach using DL, specifically the U-Net architecture. U-Net is a convolutional neural network architecture that was initially designed for biomedical image segmentation [75] and is a very reliable model for such applications.

The U-Net architecture is symmetrical, with encoder and decoder paths. The encoder consists of a sequence of convolutions followed by pooling layers that downsample the input, enabling the identification of features at multiple scales. Each convolutional layer in the encoder path uses filters to learn different aspects of the input image, while the pooling layers reduce the spatial dimensions, thus making the computation more manageable and focusing on the most

salient features (Fig. 3.17). At the network's deepest point is the bottleneck layer, which uses convolutions to analyse extremely abstract information without further downsampling. This layer is pivotal as it bridges the encoder and decoder paths, ensuring that the learned features are sufficiently abstracted to capture the intricate details required for proper segmentation. From the bottleneck layer, the features are upsampled to the data's original dimensions (Fig. 3.18). Additionally, skip links between downsampled and upsampled features increase model training speed. These skip connections allow the model to retain fine-grained details that might otherwise be lost during the downsampling process.

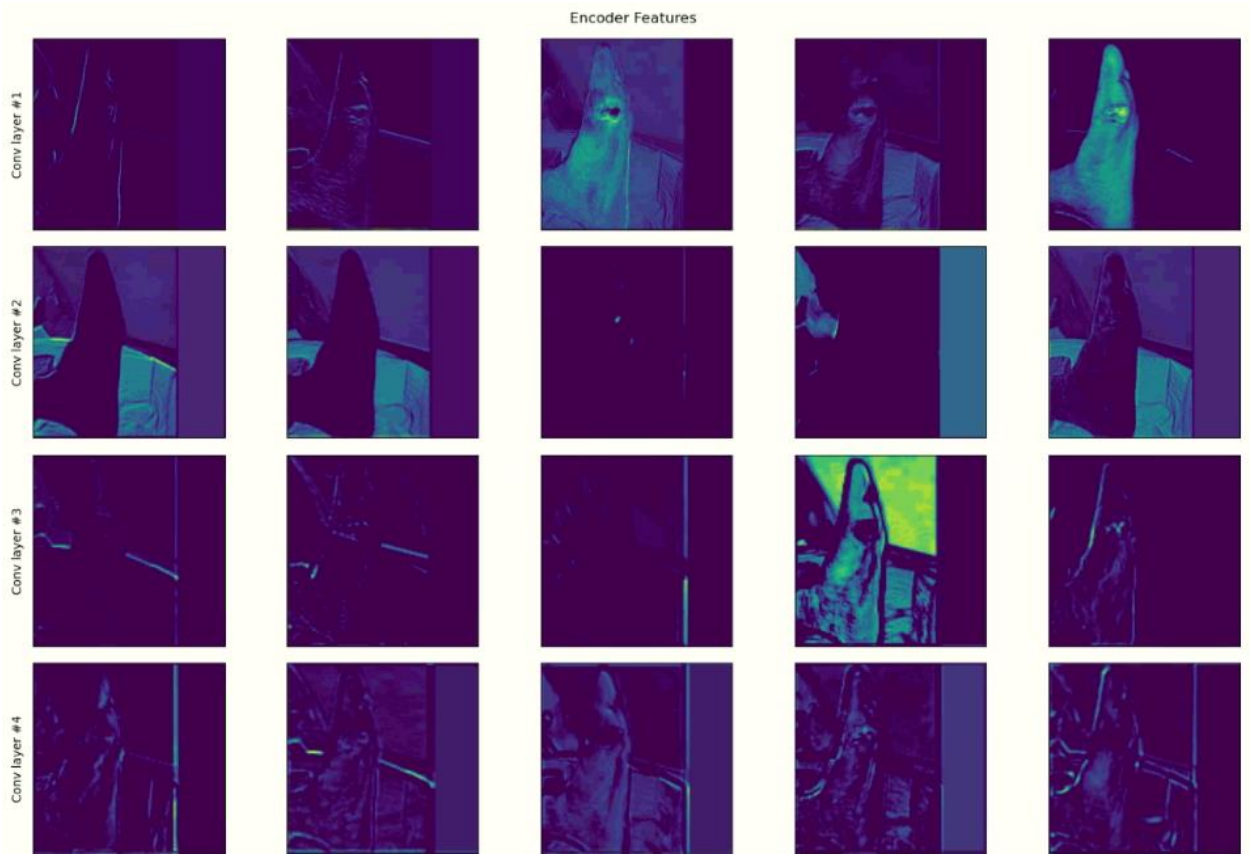


Figure 3.17 Encoder features from the first to the fourth convolution layer (rows 1-4, accordingly). We can spot how features change from low-level to high-level details: background/foreground extraction (rows 1-2), edges detection (row 4)

For our implementation of U-Net, we used 4 downsample and upsample layers with max pooling. Each layer has two convolution layers with the number of filters 32, 64, 128, and 256, respectively. Bottleneck has 256 filters with two convolution layers as well.

Figure 3.18 Decoder features highlight the output segmentation area

The U-Net model was trained on the previously mentioned AZH dataset of wound images. We added 800 images to the training set, and 50 samples were used as a validation set. We used the Dice Coefficient as the primary metric for evaluating the model's performance during training. Binary cross entropy was used as a penalty (loss) function. Adam (Adaptive Moment Estimation), an iterative optimisation approach, was used as an optimiser, with the initial learning rate 1e-5.

We trained the model for 300 epochs with a batch size of 4, which we selected experimentally: we trained the model until the loss function didn’t stop and overfitting was noticeable. The total training process took less than an hour. During the training, we used a model checkpoint to save the best weights.

Figure 3.17 illustrates the encoder features from the first to the fourth convolution layer. As the layers go deeper, the features become more abstract, capturing more complex patterns and relationships within the image. Figure 3.18 shows the decoder features, focusing on the output segmentation area. The decoder layers progressively refine the segmented regions, corroborating that the final output closely matches the ground truth mask.

3.3.1 Results and Discussion

We evaluated the performance of our U-Net model on the whole validation set. Figure 3.19 shows a direct comparison.

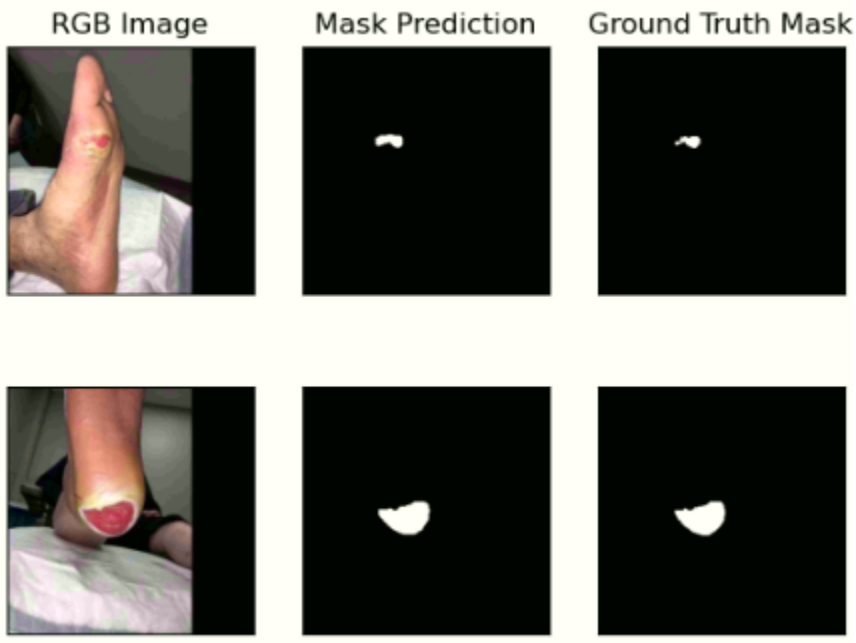


Figure 3.19 Comparison of the ground truth masks (column 3) to our predicted results (column 2)

We also tried to create a more complex situation. To prove that the model can handle occlusion, we artificially covered part of the wound. Figure 3.20 demonstrates that despite these occlusions, the model recognised the open part of the wound, indicating its ability to handle noisy objects (for example, a 3D printer).

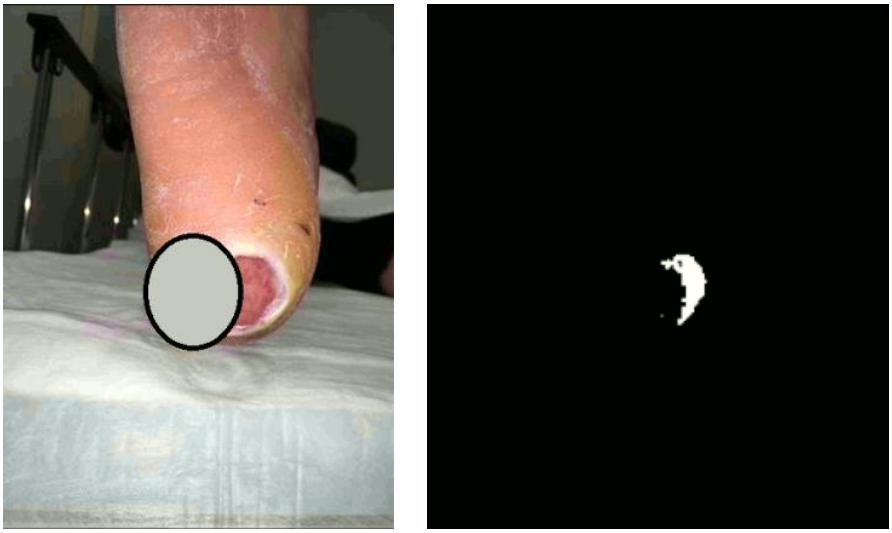


Figure 3.20 Example of handling an occlusion. The model wasn't trained with such noise, but it still can recognise the part of the wound which is open

We also measured the processing time required for segmentation. The average time taken to segment a wound mask was found to be around 9 ms, based on 50 different images (Fig. 3.21).

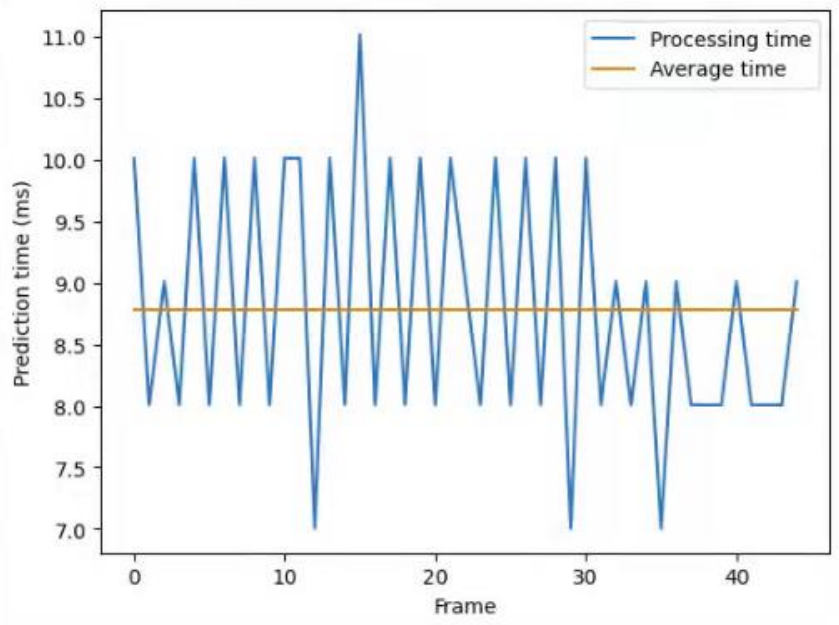


Figure 3.21 The average time to segment wound mask is 9 ms, with the highest peak of 11 ms. Accelerated on GPU (NVIDIA GeForce RTX 3080 Ti)

The model outputs probabilities (ranging from 0 to 1), indicating the likelihood that a pixel belongs to a certain class. Before calculating the score, we applied a threshold of 0.5 to exclude

low-confidence pixels. After this preprocessing step, our model's final DC score, evaluated using the same validation test, was 0.765, and the loss was 0.0126. Additional metrics showed the following results:

- Precision score (the ratio of True Positive results to total Positive results): 0.8
- Recall score (the ratio of actual Positive results to all samples that ought to have been classified as Positive): 0.77
- Accuracy score (the number of correct predictions, including both Positive and Negative, divided by the total number of predictions): 0.99

3.4 Conclusions

In this chapter, we discuss the first objective of our study. The objective was to identify a wound on the patient's body for focused 3-camera system capture. ROI segmentation reduces input frame size to a bounding box of the wound, minimising noise and focusing on the wound.

To manage synchronised video stream recording, we initialised three threads (Fig. 3.22), each running independently.

```
0 2024-06-21 19:10:47.096014
2 2024-06-21 19:10:47.348038
1 2024-06-21 19:10:47.853637
```

Figure 3.22 The synchronisation timestamp. 0-2 – camera’s ID. The timestamp of a live stream has less than a second difference between the three cameras

We reviewed two methodologies: one utilises only CV algorithms, and the other employs a DL model. The first methodology discussed proposes the use of shape matching algorithms. It detects wound contours and matches them to predefined patterns gathered from a 3D scan. Nonetheless, we couldn’t overcome the main challenge of high noise levels, which led to instability. For the second methodology, we trained a U-Net model using the available wound dataset. In contrast to the shape matching methodology, the DL model handles complex patterns and occlusions with high accuracy in segmentation.

We compared these methodologies based on accuracy (Dice coefficient metric) and processing time. The results of the first methodology returned the Dice coefficient of 0.526, with significant variation in performance depending on wound complexity. And the U-Net model achieved a Dice coefficient of 0.765. The model also leads in an average processing time of 9 ms per image, which is 30 times faster than shape matching.

In the end, the experiments demonstrated the superiority of the U-Net model for wound segmentation tasks in terms of accuracy and processing efficiency. The DL model has strong potential for real-time medical imaging applications.

4 6-DOF Motion Detection and Estimation

Summary: Tracking an object's translation along the X, Y, and Z axes, as well as its rotation about these axes (roll, pitch, and yaw) in order to detect and estimate 6-DOF motion, is the main topic of this chapter. It reviews the setup and calibration of a multi-camera system to capture 2D images and convert them into a 3D representation using triangulation. In addition, the chapter covers feature extraction and matching algorithms and demonstrates the system's functionality with tests conducted in simulated and real-world settings.

4.1 Introduction

The research goal is to detect and estimate surface motion in 6-DOF. Precise tracking assists the printer in avoiding collisions and gaps in printing material while the surface is moving or vibrating. 6-DOF motion detection involves tracking an object in three-dimensional space, capturing its translation along the X, Y, and Z axes and its rotation about these three axes: roll, pitch, and yaw (Fig. 4.1). The biggest challenge of 6-DOF motion estimation is that it involves non-linear dynamics. The main requirement for the objective is to detect even slight movements (from 300 μm).

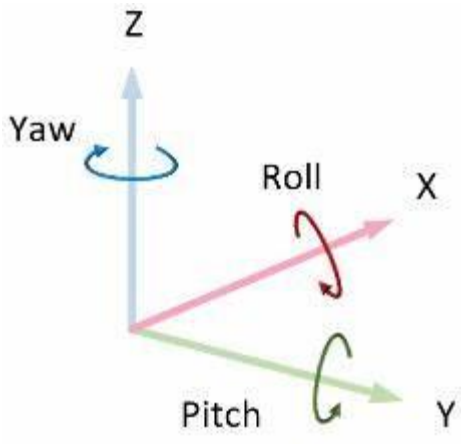


Figure 4.1 Translation and rotation vectors [76]

The camera setup design, described in the previous Chapter, excludes the possibility of detected movements when the body (part of the body with the wound) is close to an upside-down

position. The idea behind this is that the 6-DOF robot arm has its limits in flexibility, plus tissues might not engraft properly.

For this objective, we employ the same threading mechanism as the one used for the previous Chapter. Each thread generates a cropped ROI, which is subsequently processed for feature extraction and tracking. The cameras capture 2D movements, and to construct a comprehensive 3D scene, we perform triangulation. As a result, we obtain a single output consisting of XYZ displacements along with yaw, roll, and pitch angles (Fig. 4.2).

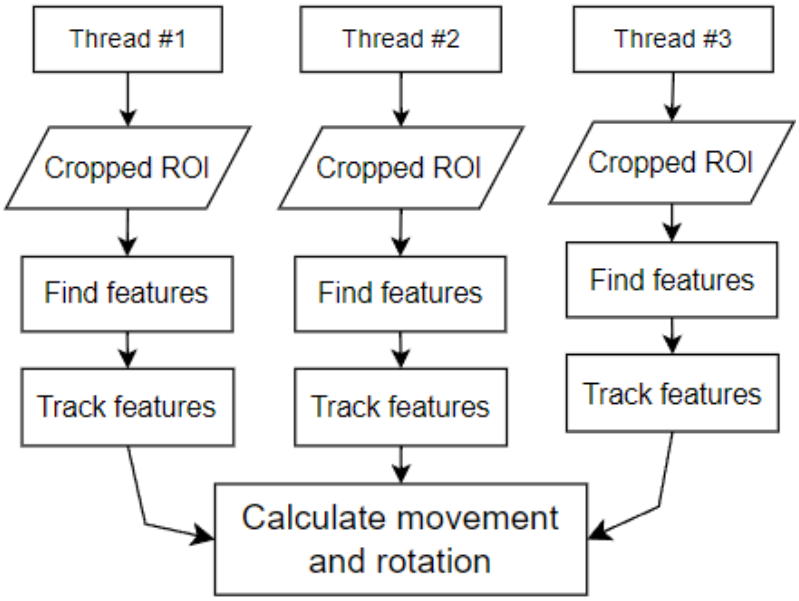


Figure 4.2 Main processing pipeline of the objective 2. The workflow is the same for each of the threads, but they process it separately. When every thread finishes, the objective returns the result based on each thread's output

Threads should be coordinated to make one output from three separate inputs. The *Event* class from the *threading* library creates a flag pointing to when a thread finishes its cycle run. The core function of the objective waits for the positive flag from each of the threads and only after starting calculations.

In this chapter, we validate the objective's output with two experiments:

- Real environment, but 1-DOF movements: using shaking table Quanser Shake Table II (earthquake simulator).
- Simulated environment, but 6-DOF motions: using the simulation in Unity3D (more familiar as a game engine).

4.2 Multi Camera System in 2D

Multiple cameras are needed to create a 3D representation because a single camera can only capture 2D images, which lack depth information. Other options, like cameras with sensors (RGBD cameras), lack precision in depth capturing. Utilising multiple cameras placed at different angles, we can gather overlapping 2D images of the same scene (Fig. 4.3). This allows us to perform triangulation, a process that uses the differences in the images captured by each camera to calculate the depth and position of objects in 3D space.

The transition from 2D pixels to 3D measurements includes camera calibration to account for intrinsic parameters, determining extrinsic parameters to relate the camera's position and orientation to the world, converting pixel coordinates to camera coordinates, and then mapping these to world coordinates using rotation and translation vectors. Finally, a scaling factor is applied to convert the units to millimetres so the robot hand can navigate using a common coordinate system.

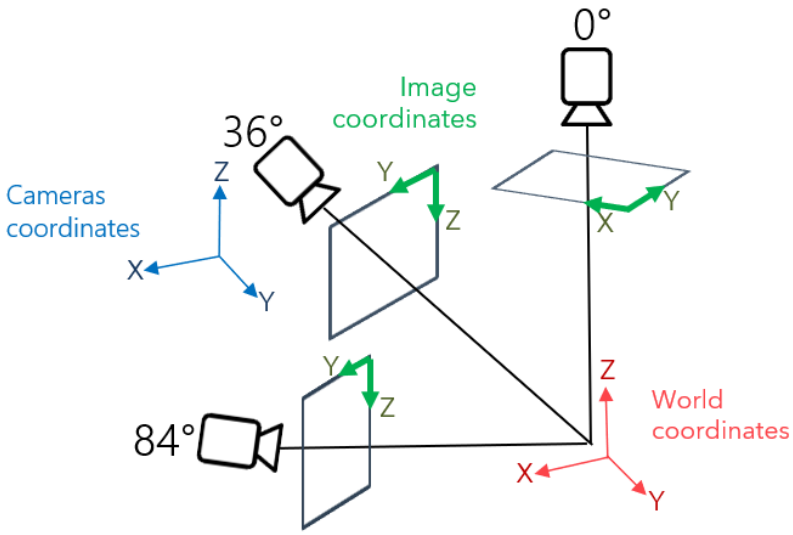


Figure 4.3 Conversion of the camera (pixels) to world (meters) coordinate system. Image coordinates are labelled respectively to world coordinates

Extrinsic parameters define the camera's position and orientation in the world coordinate system. Extrinsic parameters can be represented as a 3x4 matrix, where a 3D vector describes the camera position (Equation 4.1) and a 3x3 matrix (Equation 4.2) – camera rotation relative to the world.

$$t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{4.1}$$

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{4.2}$$

Each element r_{ij} represents the state of the angle between the i-th axis of the world coordinate system and the j-th axis of the camera coordinate system. r_{1j} represents how the world's X-axis projects onto the camera's XYZ, r_{2j} represents the world's Y-axis and r_{3j} – represents the world's Z-axis.

The rotation matrix R can be decomposed to extract the roll (θ_x), pitch (θ_y), and yaw (θ_z) angles of the camera (Equation 4.3).

$$\theta_y = \arctan\left(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}\right);$$
$$\theta_x = \arctan\left(\frac{r_{32}}{\cos\theta_y}, \frac{r_{33}}{\cos\theta_y}\right);\ \theta_z = \arctan\left(\frac{r_{21}}{\cos\theta_y}, \frac{r_{11}}{\cos\theta_y}\right) \tag{4.3}$$

Intrinsic parameters are the internal characteristics of the camera that affect how it captures images. The camera matrix can be represented as a 3x3 matrix (Equation 4.4) and a set of distortion coefficients (Equation 4.5).

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.4}$$

Where f is a focal length, c is an optical centre.

$$dist = \{k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3\} \tag{4.5}$$

Where k is the radial distortion coefficient to correct for bulging or pinching effects in the image, and p is the tangential distortion coefficient to fix the skewing effect.

The full transformation from pixel to world coordinates can be expressed as the use of the intrinsic and extrinsic parameters to map a point from the image plane through the camera system and out into the world coordinate system (Equation 4.6).

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = R^{-1} \cdot \left(K^{-1} \cdot \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} - t \right) \tag{4.6}$$

Where (X, Y, Z) – the coordinates of a 3D point in the world coordinate space (with the fixed Z coordinate), and (u, v) – the projection point coordinates in pixel. While the calibration considers the Z-axis as fixed, we circumvent this limitation with our setup. Two cameras are placed perpendicularly to each other: the top camera captures XY-axes in its plane as well as in the world plane, while the bottom camera captures ZY-axes in world coordinates (Fig. 4.3). To clarify, since the cameras are placed perpendicularly, the bottom camera's Y-axis corresponds to the Z-axis of the top camera (and X-axis of the bottom camera corresponds to Y-axis of the top one). Therefore, the information captured by these two cameras combined to reconstruct the full 3D position in real-world units of any point in the scene.

To calibrate all our cameras, we used a chessboard pattern (Fig. 4.4). We applied an OpenCV method *calibrateCamera()* to each of the cameras to find camera matrices, distortion coefficients, rotation and translation vectors.

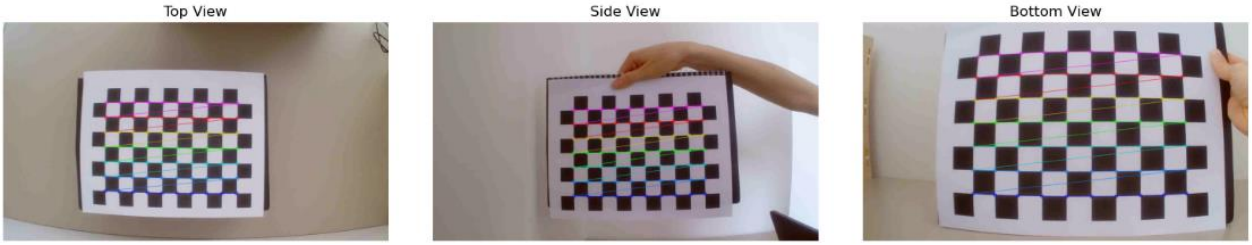


Figure 4.4 Calibration process using chessboard pattern

The resulting values are shown in Table 4.1. These values are used later to recalculate a new camera matrix using another OpenCV method, getOptimalNewCameraMatrix(); in this function, we set the alpha parameter to 1 to retain all pixels as we crop frames to get ROI afterwards. The next stage is to undistort the image; again, it is done with the OpenCV *undistort()* method. This method is applied to the entire frame (not yet cropped) with the new matrix and distortion coefficients. Finally, we crop the ROI on the undistorted image, resulting in an image with no fisheye effect.

Top camera	Camera matrix	$\begin{bmatrix} 2155.93 & 0 & 749.77 \\ 0 & 2125.24 & 318.26 \\ 0 & 0 & 1 \end{bmatrix}$
	Distortion coefficients	$[-1.5 \quad -0.65 \quad 0.05 \quad -0.02 \quad 13.77]$
Side camera	Camera matrix	$\begin{bmatrix} 980.82 & 0 & 601.57 \\ 0 & 1002.93 & 445.87 \\ 0 & 0 & 1 \end{bmatrix}$
	Distortion coefficients	$[-0.08 \quad -1.31 \quad -0.04 \quad 0.03 \quad 1.53]$
Bottom camera	Camera matrix	$\begin{bmatrix} 5258.69 & 0 & 703.83 \\ 0 & 5800.13 & 302.84 \\ 0 & 0 & 1 \end{bmatrix}$
	Distortion coefficients	$[-10.81 \quad 114.43 \quad 0.13 \quad 0.05 \quad 5751.0]$

Table 4.1 Cameras' intrinsic parameters

We can compare how images changed before (Fig. 4.4) and after (Fig. 4.5) undistortion.

Figure 4.5 Undistorted images. The new dimension is written in the title

Undistorted images changed the dimensions of the original images (720x1280) and added black pixels at the image corners (Fig. 4.5 column 2).

We can determine how accurate the calibration was by looking at the reprojection error. Reprojection error calculates the Euclidean distance between the real points seen in the picture and the points projected by the derived camera parameters. A chessboard pattern is designed with precise dimensions, meaning the spacing and size of the corners of the squares in a board are controlled and known. Our calibration error is low – 0.04 pixels.

The final step is to determine the length of each pixel in the real-world unit (millimetres). For this, we again use a chessboard pattern, but now as a real-world size reference. Each square side is 25 mm, but the distance between their corners is measured in pixels. To calculate the scale factor, we divide real distance by pixel distance. Because of the different distance between the camera and the object, the scale factor is different for each camera:

- Top camera: 0.508 mm/px
- Side camera: 0.51 mm/px
- Bottom camera: 0.33 mm/px

4.3 Feature Extraction

Feature extraction involves identifying key points in the frame that can be used for tracking. In this section, we review two methods for feature extraction.

The first method is an OpenCV implementation – *goodFeaturesToTrack()*. It first identifies distinct pixel blocks on the target object. A video stream locates previously identified blocks at their new coordinates (Fig. 4.6). It is designed to find the most distinguishable corners in the image using the Shi-Tomasi corner detection algorithm. The method can be adjusted with several parameters: the maximum number of corners to return, the corner quality threshold, and the minimum possible Euclidean distance between the returned corners. While this method is very fast, the processing time to find 100 features takes only 42.5 µs. This method is not completely scale-invariant, and this is the main requirement for 3D tracking.

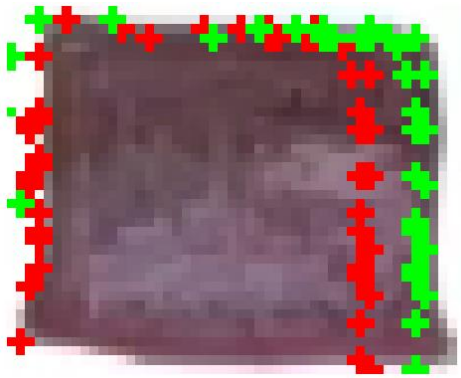


Figure 4.6 Tracking 30 features. Red crosses – feature coordinates on a previous frame, green crosses – a new position. Herein, there is a displacement to the right

SIFT is another powerful method included in OpenCV for feature extraction, particularly known for its robustness to changes in scale, rotation, and illumination (Fig. 4.7). Similar to the previous method, it detects key points in the image and generates descriptors. Descriptors are unique vectors that describe the local image gradients around each key point detected by SIFT. To avoid illumination dependency, the SIFT algorithm searches for local extrema in a series of Gaussian-blurred images and assigns a consistent orientation to each key point based on local image gradient directions to keep invariance to scale and rotation.

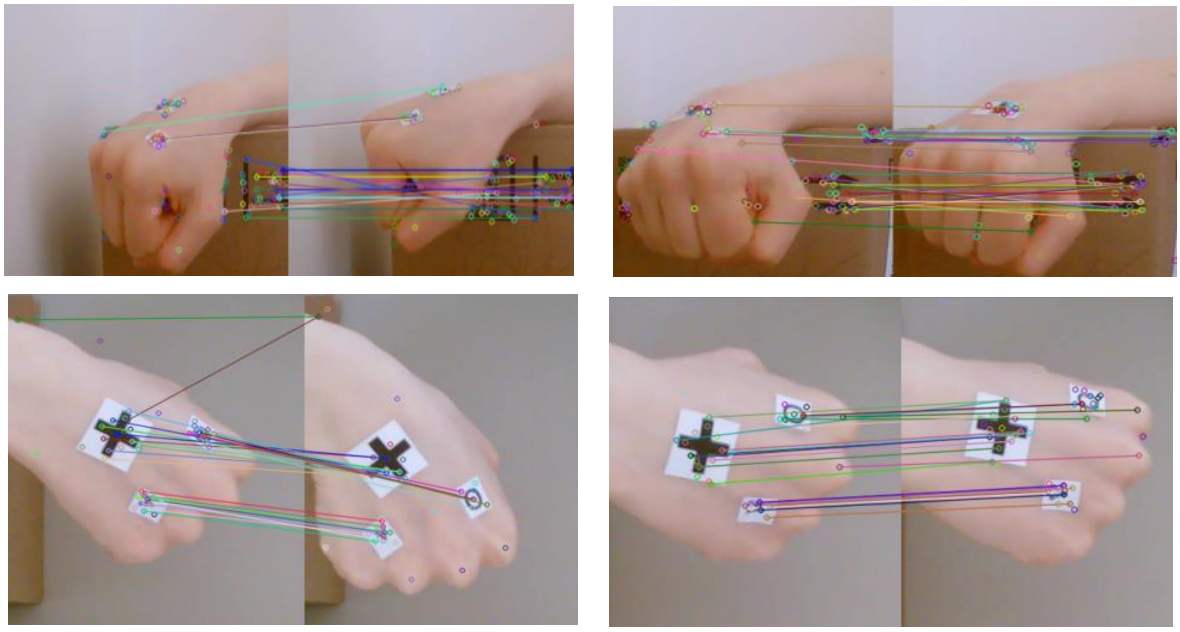


Figure 4.7 Matched features to track. Top row – point of view from the side camera, bottom row - top view camera. The first column is yaw rotation clockwise, and the second column is rolled counterclockwise. Features are drawn and connected using the *drawMatchesKnn()* method

To detect a list of features on the image, we create a SIFT instance with the number of features to extract. To find key points and descriptors in a single step, we use *sift.detectAndCompute()* function. In the first two video frames, we need to extract key points and descriptors for both frames. But at the next iteration, we calculate them only for the new frame and reuse the previous coordinates.

4.4 Feature Tracking

Feature tracking is used to follow predefined features within a series of frames in a video. Tracking provides continuous monitoring and analysis of specific points on the body or in the wound, making sure that tracking movements are precise even as the position and orientation of the surface change.

In our tracking system, we evaluated the Lucas-Kanade Optical Flow method, implemented through the *calcOpticalFlowPyrLK()* function in the OpenCV library. This method calculates the optical flow for a sparse feature set using a pyramidal approach. It works on the assumption that the flow is essentially constant in a local neighbourhood of the pixel under consideration. The algorithm calculates the motion of specific features by solving basic optical flow equations (Equation 4.7-4.9) for each pixel in the neighbourhood. This function returns an array of the new positions detected by the input feature points in the second image.

$$I(x, y, t) = I(x + dx, y + dy, t + dt) \tag{4.7}$$

$$f_x u + f_y v + f_t = 0 \tag{4.8}$$

$$f_x = \frac{\partial f}{\partial x};\ f_y = \frac{\partial f}{\partial y};\ u = \frac{\partial x}{\partial t};\ v = \frac{\partial y}{\partial t} \tag{4.9}$$

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{z_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i f_{x_i} f_{t_i} \\ -\sum_i f_{y_i} f_{t_i} \end{bmatrix} \tag{4.10}$$

Lucas-Kanade method (Equation 4.10) solves the problem of unknown (u, v), its solution is obtained with the least square fit method. f_x and f_y are image gradients, and f_t is the gradient along time.

To locate previously detected features in new frames, we can adopt two feature matching (Fig. 4.7) methods: *FlannBasedMatcher()* or *BFMatcher()*. The difference between the FLANN (Fast Library for Approximate Nearest Neighbors) and BF (Brute Force) matchers is that the first one provides an approximate solution to the nearest neighbour search problem (Equation 4.11) and is particularly efficient for a large number of features (more than 1,000), on the other hand, the BF matcher, exhaustively searches for the best match for each feature, providing accuracy but at a higher computational cost.

$$\widehat{q_i} = \arg\min_{q_j} D(d_{p_i}, d_{q_j}) \tag{4.11}$$

In Equation 4.11, feature matching algorithms find the best match by minimizing the Euclidean distance metric D between given a set of descriptors d_p from the first image and a set of descriptors d_q from the second image. $\widehat{q_i}$ represents the best match for every feature in the descriptors from the first image set.

The Kalman filter is a recursive algorithm used to estimate the state of a dynamic system using a series of noisy measurements. It operates in a two-step process: prediction (before observation (Equation 4.12-4.13)) and update (observation point (Equation 4.14-4.18). It updates the state based on new measurements. Kalman filter predicts the object's new position in the next frame based on the observations. Thus, it can help with feature tracking through the occlusion.

$$X_t^- = A_{t-1}X_{t-1} + B_tU_t \tag{4.12}$$

$$P_t^- = A_{t-1}P_{t-1}A_{t-1}^T + Q_{t-1} \tag{4.13}$$

Where X_t^- – predicted mean and P_t^- – predicted covariance of the state at time t.

$$V_t = Y_t - H_tX_t^- \tag{4.14}$$

$$S_t = H_tP_t^-H_t^T + R_t \tag{4.15}$$

$$K_t = P_t^-H_t^TS_t^{-1} \tag{4.16}$$

$$X_t = X_t^- + K_tV_t \tag{4.17}$$

$$P_t = P_t^- - K_tS_tK_t^T \tag{4.18}$$

Where X – estimated mean and P – estimated covariance. Y – mean measurement, V – measurement residual, S – measurement prediction covariance, K – Kalman filter gain, which indicates the number of corrections needed for the predictions.

In short, all the approaches return new positions in the second frame of the detected features from the first frame. To find an actual displacement difference between previous and new coordinates, we subtract new feature placements from the previous locations one by one per axis and take a median of the result. In the aftermath, we reckon an object's velocity between the frames. This velocity is a new observation for the Kalman filter and is used to correct its predictions.

4.5 Rotation Estimation

The perspective transformation, or homography, between two frames has to be ascertained before we can estimate the rotation of an object. A homography (Equation 4.19) is a mathematical transformation that takes perspective shifts into account when mapping points between frames (Equation 4.20). We use the RANSAC algorithm in conjunction with the OpenCV *findHomography()* method to implement this.

$$\mathrm{H} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{4.19}$$

$$\begin{bmatrix} x_i' \\ y_i' \\ 1 \end{bmatrix} = \mathrm{H} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{4.20}$$

Where $(x_i', y_i')$are the image coordinates of one of the features in one image and (x_i, y_i) are from the second image.

By finding feature point matching between the two frames, the *findHomography()* method computes the homography matrix. Through iteratively choosing random subsets of correspondences, estimating the transformation, and assessing its consistency across all points, RANSAC (Random Sample Consensus) is utilised to robustly estimate the homography. This algorithm aids in identifying the best-fit homography and discarding outliers.

We can deconstruct the homography [77] to recover the rotational component once it has been calculated. In order to isolate and quantify the rotation of the object across the frames, we must first separate the translational and rotational components of the transformation matrix. By calculating the arctangent of the ratio between the matrix's elements, we find the rotation angle in radians. To find the rotation point in a 2D plane, we use the components of the translation vector and the cosine and sine of the rotation angle (Equation 4.21-4.22).

$$P_x = \frac{t_x}{1 - \cos\theta} \tag{4.21}$$

$$P_y = \frac{t_y}{\sin\theta} \tag{4.22}$$

Where P_x, P_y is a rotation point at the specified coordinate, t_x, t_y – translation vector, and θ is the rotation angle.

4.6 Experiments and Results

To demonstrate the proposed methodology, we tested it in different environments. We will start with the simple 2D estimations of X-axis movements, review our algorithm's estimations of simulated 6-DOF movements, and, in the end, demonstrate rotation estimations with the real camera system setup.

The first experiment setup involves the following items (Fig. 4.8):

- Shaking table: Quanser Shake Table II (earthquake simulator).
- One web camera on top of the table: 29.91 FPS, 1280x720.
- Arbitrary ceiling lightning.
- Hand sketch of a free form as a target object.

We captured all the possible frame timestamps with respect to the camera's characteristics, and one frame equals 33.43 ms timestamp.

The shaking table can execute only X-axis vibrations. For our experiments, we set it with two frequencies: 0.1 Hz (or 0.08 mm distance per frame) and 0.5 Hz (0.47 mm per frame). Also, the table logs its position state over time, which we used as validation data.

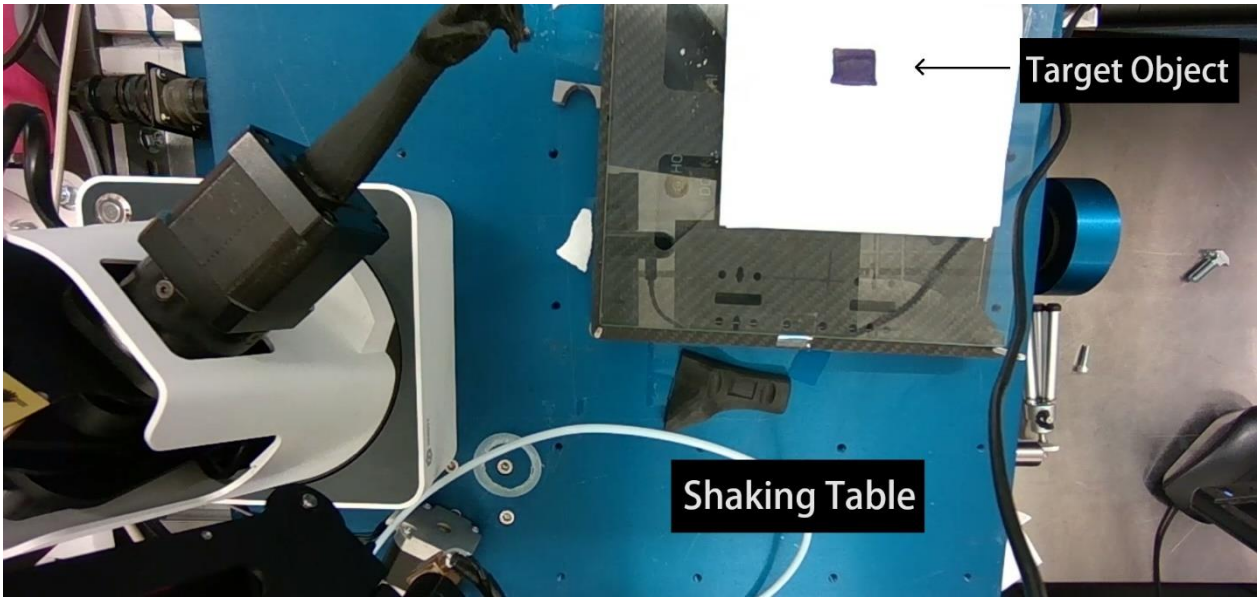
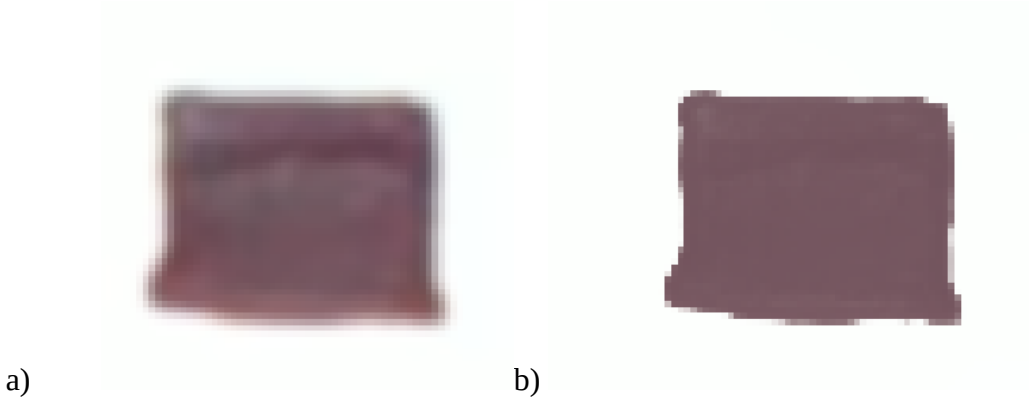


Figure 4.8 2D movements setup: shaking table, the camera is strictly on top above the target object

Before we calculate the movements, we preprocess the input video with the algorithm from Chapter 3. When we got the ROI, we decided to blur it (Fig. 4.9) because the drawing was reflective. The bilateral filter showed the best result because it removed all the reflections but kept the contour's shape.



a) b)

Figure 4.9 Comparison of Gaussian blur (a) and bilateral filter (b). The second one removes reflections and keeps the same shape, while the Gaussian method blurs the contour

To estimate movements, we used *goodFeaturesToTrack()* along with *calcOpticalFlowPyrLK()*. We set a maximum of 30 features to detect, with a quality above 0.3 and a minimal distance of 1 px for the feature extraction method configuration. That helped to detect both sides of the drawing and not confuse them later, but at the same time, we kept a low-quality level to ensure that at least some feature blocks would be selected.

From the Optical Flow, we received new feature point locations, from which we subtracted previous locations and took a mean value. If the value is negative, it means that the object moved to the left; if it is positive – it is to the right.

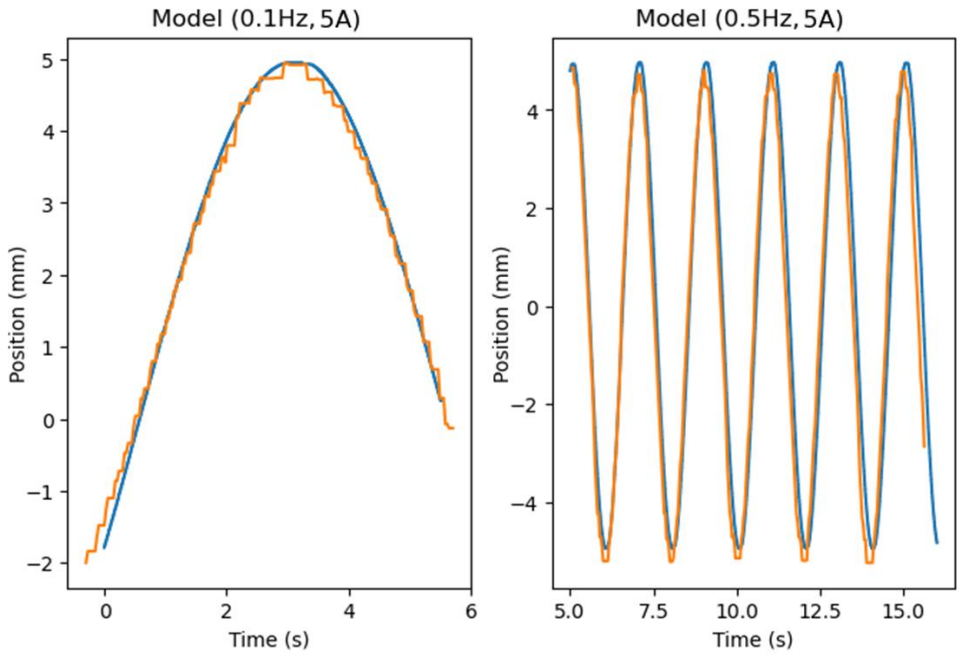


Figure 4.10 Comparison of ground truth data from the shaking table (blue) to our results (orange)

There is some imprecision visible in Figure 4.10. One of the reasons for that is that the shaking table reports non-linear movements for a small time range (0.03-0.04 s) (Fig. 4.10). The used camera captured one frame per 0.03 s, which means each frame could fall in the "non-linear gap", and it could cause some curves in the resulting plot (Fig. 4.11). The "More linear" view of the values of ground truth data is a result of a larger amount of data since the table reports movements for every 0.0005 s.

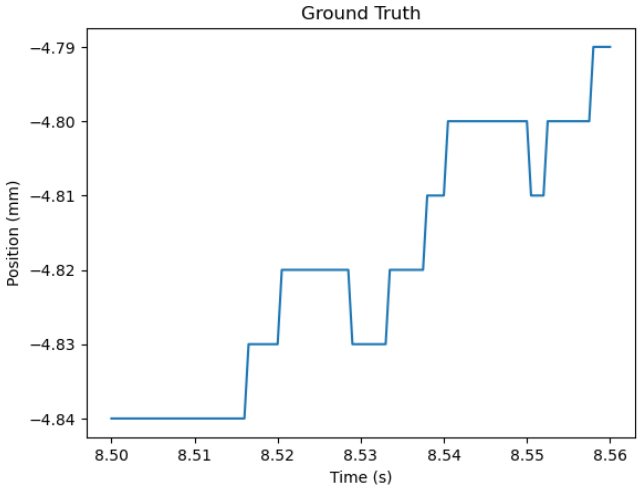


Figure 4.11 Non-linearity of ground truth data

For our second experiment, we created a virtual scene in Unity3D, where we reproduced our experimental design (Fig. 4.12):

- 3 cameras placed at the previously discussed angles: 60 FPS, 1920x1080.
- Even natural lightning. It didn't cast shadows but caused some minor reflections.
- The main object was a cube with distinctive features on one side that moved and rotated with the constant speed of 0.3, both at the same time.

The virtual scene has enabled a log with all recorded translation and rotation state changes of the cube. Unity3D returns the object's state as two vectors: 3D position vector (XYZ) and rotation quaternion. To match rotation states with our output format, we converted quaternions to Euler angles. The recorded states were written with the Unity3D time frequency, which is equal to 60 FPS. Using these records, we validated our results. To get input videos from the simulation, we recorded each of the cameras' output using the Unity3D assets plugin. Each camera was recorded separately. For faster processing, we selected every 3rd frame; same for the Unity3D log, we used only every 3rd row of data.

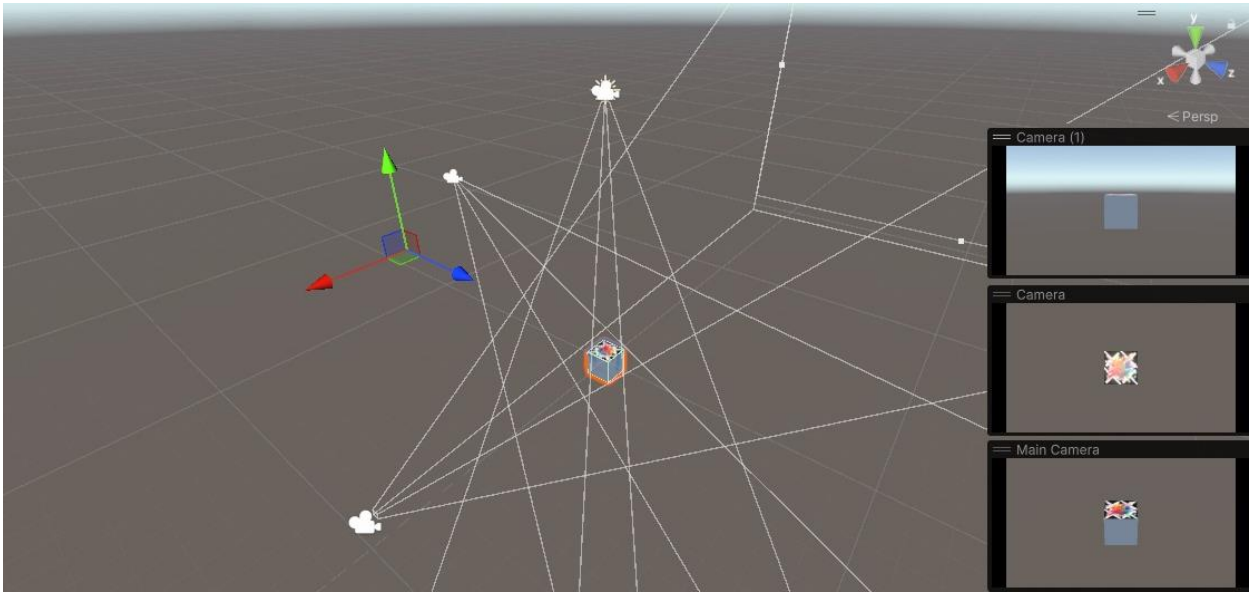


Figure 4.12 Setup simulation in Unity3D. Three cameras are placed at the top, side, and bottom (on the right part of the screenshot, as follows: Camera, Main Camera, Camera (1)). A cube is the main object, which moves with a constant speed in all 6-DOF simultaneously

Since it is a simulated environment, we can’t get real-world distance measurements. Unity3D has its own coordinate system. For example, the cube edge is 1 unit. We used it as a reference for the scale factor calculation. We got that 282 px equals 1 unit.

In our core algorithm, we used SIFT detector and FLANN to perform feature matching. SIFT was configured to find 100 features. To configure the FLANN matcher, we updated its index and search parameters. For the index parameter, we used *FLANN_INDEX_KDTREE* with 4 trees, which means it utilises 4 kd-trees (data structure of k-dimensional space). The search parameter was set to perform only 10 feature checks per iteration. The next step was to return the best 2 matches using *Flann.knnMatch()*, meaning we can apply Lowe's ratio test to filter good matches. Next, we estimate rotation differences using the matched key points, which we pass next to calculate the transformation matrix using RANSAC. The result of rotation estimations is in radians, which we convert to degrees to match with Unity3D coordinates. To find the 2D displacements, we apply the median difference between source and destination points.

As we know, cameras capture only 2D images. To explain 3D movements, we used some cameras of our setup to detect XZ and YZ movements and roll, yaw, and pitch rotations separately in different planes. Movements along the X-axis were detected by the top camera, while movements along the Y and Z-axes were recorded by the bottom camera. Likewise, rotations around the X-axis were captured by the top camera in combination with rotations around the Y-axis being monitored by the side camera and rotations around the X-axis by the bottom camera (Fig. 4.13).

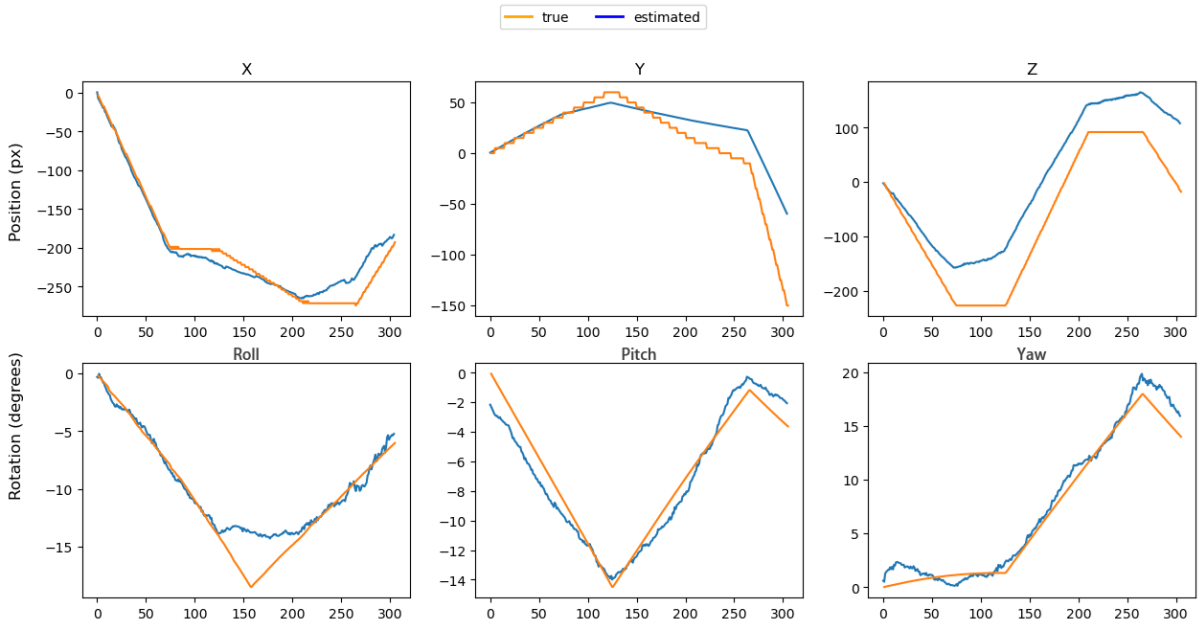


Figure 4.13 Estimated movement results. All the plots draw cumulative movements. Orange line – true values, blue – estimated values. Columns correspond to coordinates (XYZ). Top row – position difference, bottom row - rotation

From Figure 4.13, the estimations for both position and rotation are generally accurate, follow true movement tendency, and the estimated lines follow the true data closely in most cases. As the results above are cumulated values, we explored frame-by-frame errors to examine the difference between the true and estimated values (Fig. 4.14).

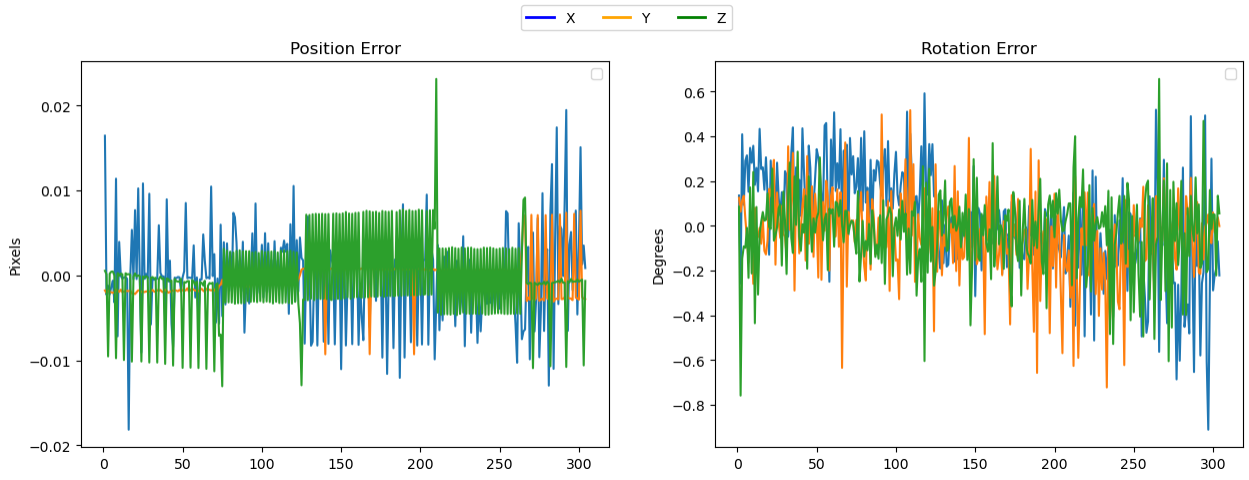


Figure 4.14 Difference between ground truth and estimations. Left plot – position difference, right one – rotation. For positions. The absolute difference doesn't exceed 0.02 px for position error and 0.8 degrees for rotation

Non-cumulative discrepancies between the true and estimated movements show minor deviations of approximately 0.02 px among XYZ transition and up to 1 degree for rotation error. Additionally, we validated the precision of our algorithm using several error metrics. The transition errors and rotation errors were quantified as follows:

- Mean absolute error:
    - X: 0.0038
    - Y: 0.0015
    - Z: 0.0036
- RMSE translation Error:
    - X: 0.0052
    - Y: 0.0021
    - Z: 0.0048
- Mean Rotation Error (Geodesic Distance): 0.006
- Quaternion Distances: 0.006

In our first experiment, we demonstrated our system's capability to estimate movements within a real-world environment. In the current experiment, our aim is to evaluate the system's ability to estimate real-world roll and yaw rotations. This is executed by utilising two web cameras positioned at the top and side perspectives. To conduct this evaluation, I used my wrist, which is rotating in 4-DOF (roll+yaw and natural sliding on the stand). I attached random features to my wrist, and the cameras captured the wrist’s rotations based on the tracking of these features.

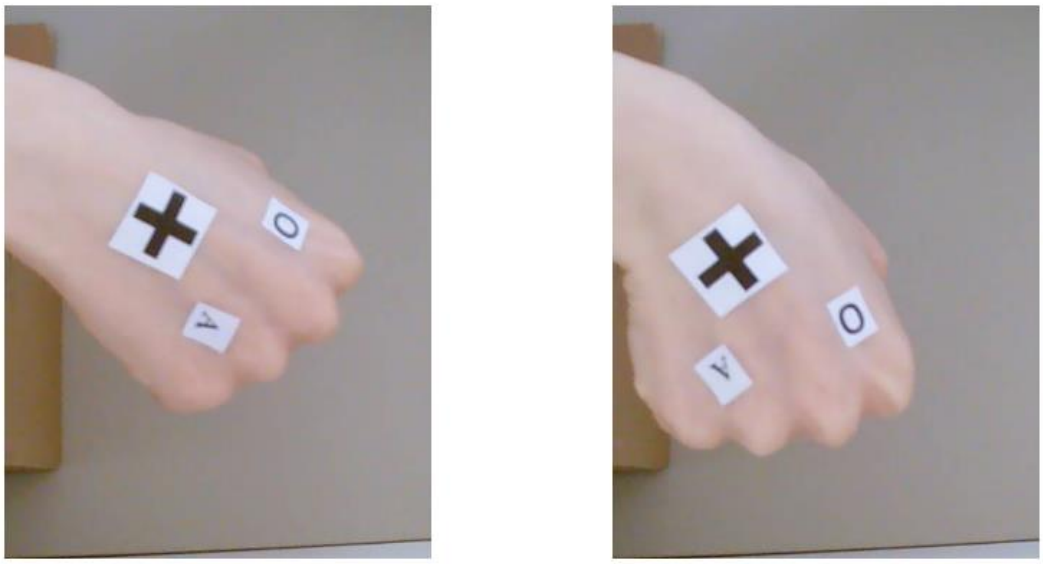


Figure 4.15 Real environment example of the yaw rotation of the hand. For this particular test, the system estimated yaw at 20.88 degrees (clockwise)

With the input of images shown in Figure 4.15, where the left image is the source image, and the right one is the destination, our algorithm detected a 20.88 degrees rotation, in contrast to 25 degrees measured manually using a digital protractor. A positive value indicates the clockwise rotation, while a negative is counterclockwise.

Figure 4.16 Another real environment example is from the side camera's point of view. Here, the system estimated roll at -25.87 degrees (counterclockwise)

The input of Figure 4.16 returned a detected -25.87 degrees counterclockwise roll rotation when manual measurements showed approximately -16 degrees. We explain such a severe difference because the majority of detected features were extracted from a box's drawings rather than from the hand.

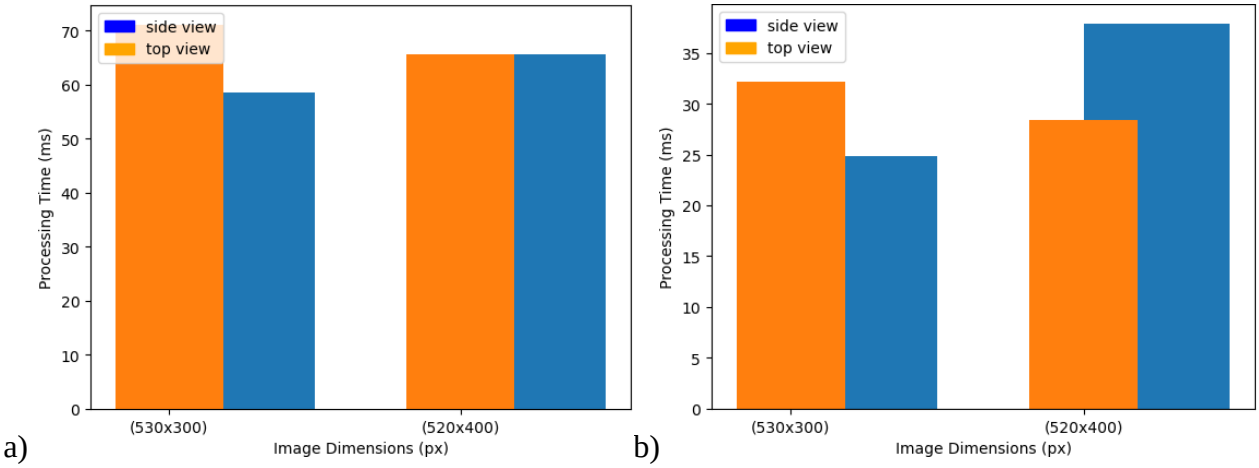


Figure 4.17 Processing time for the complete pipeline: feature detection, tracking, and position and rotation estimation. (a) first run – finding two sets of key points, (b) second run – reusing the previous key points set

The final part of our analysis is to measure the processing time for our algorithm. Figure 4.17 provides the necessary information; from it, we can see that the initial run, when we calculate key points and descriptors for both images, is being processed twice slower but doesn't exceed more than 70 ms per iteration.

4.7 Conclusions

In this chapter, we introduce the concept of 6-DOF motion detection and estimation. This involves determining an object's motion in three-dimensional space, accounting for translations along the XYZ axes and rotations around these axes.

The current camera setup restricts movement detection when the body is in an unexpected position because it can lead to the unpredictable behaviour of the robot or printed material. Camera calibration not only flattens images but also provides real-world distance measurements.

On the software part, each thread from the previous Chapter handles an ROI for feature extraction and tracking. Threads' wait functionality guarantees that main computations begin only

after each thread has finished its cycle. In the core part, triangulation constructs a 3D scene from 2D camera captures, resulting in estimated XYZ displacements and rotational angles.

Our tests demonstrated that the optical flow approach performs well for simple movements in an actual setting. The algorithm, which combines SIFT and FLANN, demonstrated accurate feature tracking in a Unity3D-simulated environment, with only small translation and rotation errors. Because of flaws in feature extraction, rotation estimation in a real-world setting displayed imprecision. The algorithm run takes no more than 70 ms to process in total, including key point and descriptor calculations.

5 Printing Trajectory Generation

Summary: This chapter examines the process of generating and optimising 3D printing trajectories for use in bioprinting applications. It covers the creation of initial trajectories, the strategy to update them in response to environmental changes, and the system's ability to dynamically alter to guarantee precise and efficient deposition of biomaterials. The analysis of different trajectory pattern orientations to identify the best option for reducing printing time without sacrificing accuracy completes the chapter.

5.1 Introduction

Another necessary objective is to initialise the shortest 3D path for printing biomaterials to deliver the surgery as fast as possible, keeping the same accuracy.

Printing trajectory is a path of a 3D bioprinter's nozzle to fill up the wound. The wound itself could be viewed as a layered object, as skin has three main layers: hypodermis, dermis, and epidermis [78]. This leads us to a horizontal slicing design for the printing path (Fig. 5.1).

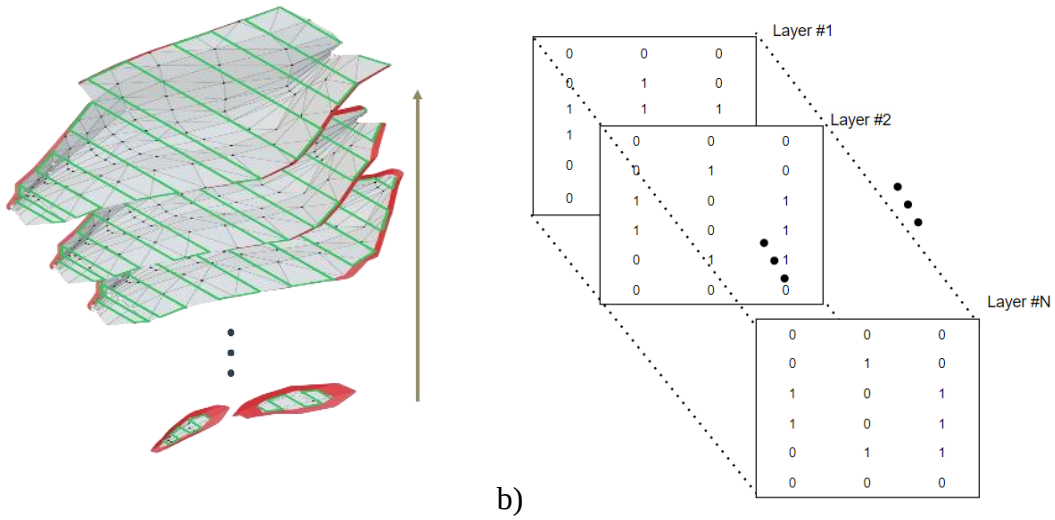


Figure 5.1 A horizontally sliced 3D wound model. Visual (a) and approximated numerical representation (b)

To follow any surface movement during printing, the trajectory should be modified immediately, considering all the orientation and rotation changes. Besides the above, the system

must not blindly follow the generated trajectory; it should know the printing progress – what part/layer has been printed already, where it is now, and what will be printed next.

To consolidate this concern, we impose a new trajectory representation by using a checkpoint system. Every checkpoint can be viewed as a printing path descriptor. It can contain not only position and orientation coordinates but also required printing material characteristics for the specific spot in the wound, along with the desired printing speed: lower for curved trajectory, higher for more straight paths. By using a checkpoint system, the system can keep in memory the material printed at this moment, what material should be printed next, and the last printed point location in case the robot stops extrusion due to some specific environmental conditions. Every checkpoint is represented as at least a 9D vector: [X, Y, Z, α, β, γ, speed, material, temperature], with the flexibility to append the additional required information. While checkpoints can have an N dimension, only the first 6 elements, position and rotation coordinates, are adjustable during surface movements.

Furthermore, the trajectory should be updated in every frame due to the slightest movements. This means that we cannot yield computational power every time we detect movements. Such a scenario would lead to system lags when the environment changes faster than we capture those changes. For the current system capabilities (cameras' resolution, connector bandwidth, and processing power), the lag reaches ~180ms (5 frames), but with the system expansion, this time might increase drastically. To resolve this issue, we created a separate processing thread to calculate trajectory transformation.

5.2 Initial Trajectory Generation

We use a 3D mesh obtained from the 3D scanner to create the initial trajectory. However, Python, our system prototype programming language, cannot directly handle a 3D mesh. Therefore, we must first convert the mesh into a numerical array with dimensions Nx3, where N is the total number of points and 3 is the XYZ coordinates. To get the numerical representation, we process the input mesh using the PyVista library to extract the points. Following this, we slice the mesh layer by layer to generate a separate trajectory for each of the layers.

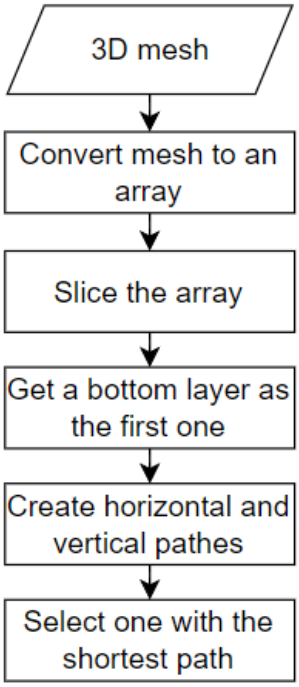


Figure 5.2 Diagram of the initialisation logic

We split the vertices into horizontal layers based on depth information. The thickness of these layers is determined by the properties of the biomaterial since each tissue layer has a unique thickness. Our algorithm reserves a controlled variable for the thickness of each layer and adjusts it independently, allowing for customisation based on specific requirements.

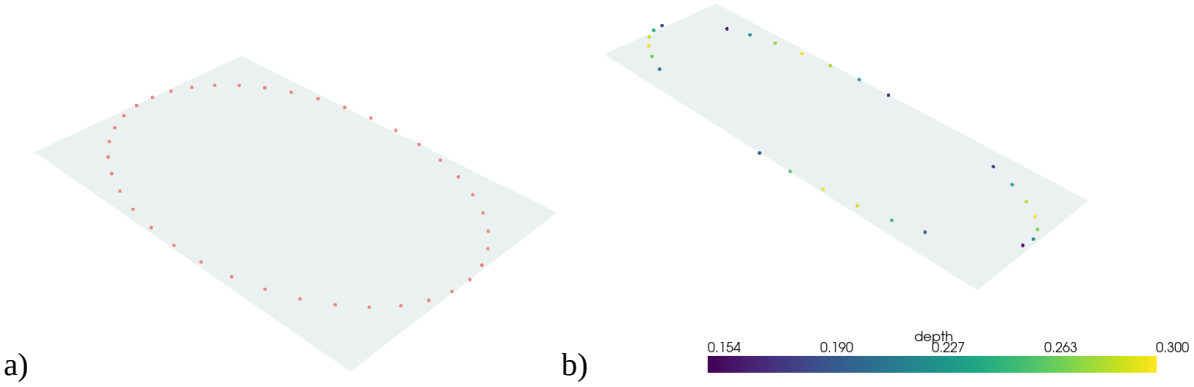


Figure 5.3 Based on the previously shown 3D shape in Chapter 3 (Fig. 3.7), we divided it into 4 layers. (a) the bottom layer, (b) the top layer, including the depth map

Before proceeding further, we should address the challenge of mapping 3D model coordinates to world coordinates. 3D model coordinates are positioned relatively at the starting point O (0,0,0). To update these coordinates, we find the centre (centroid) of the segmented wound mask with the image moments (when pixel weight equals its intensity). With performed calibration, we estimate the X and Y world coordinates of the mask centroid. Next, we apply the centroid coordinates to the rest of the 3D model points. The bottom layer's Z value is assigned to be 0 in the printer's coordinates, representing the base level of the printing process, as the printer starts at this lowest possible point and cannot print below this level.

After the splitting and mapping to real world coordinates, we pave the printing path from the bottom layer to the top layer (Fig. 5.3). The printing begins at the bottom layer. Once a complete circle is printed, we increase the value of the Z coordinate by the thickness variable of the next layer to move up. This process is repeated layer by layer until the final layer is reached.

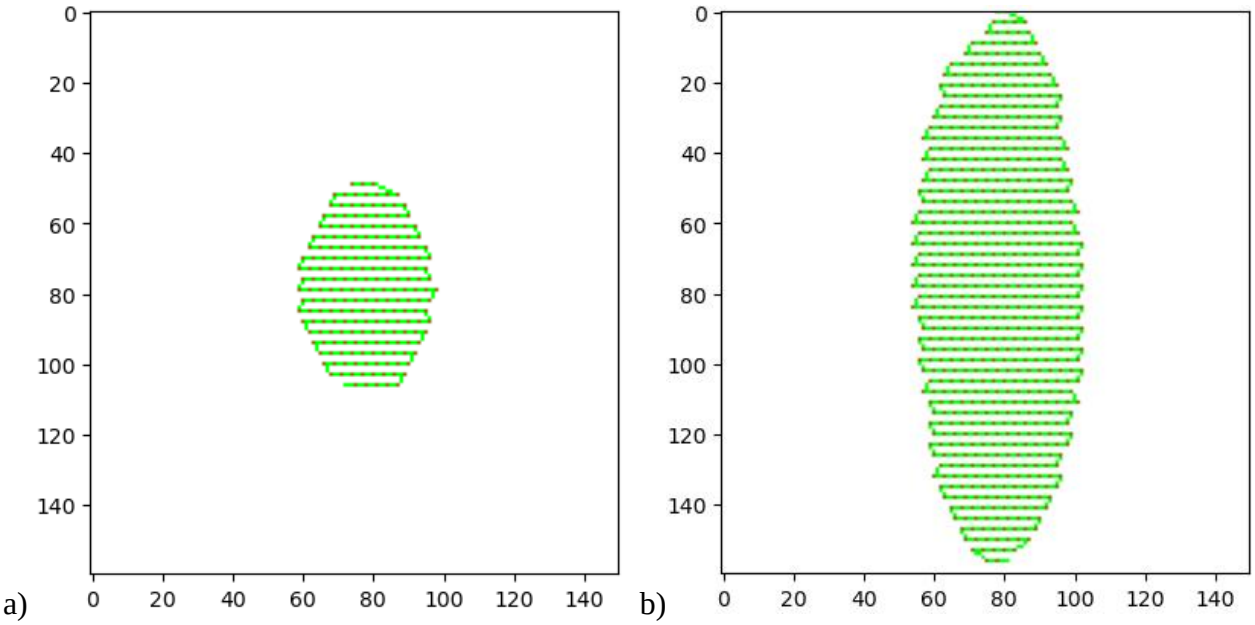


Figure 5.4 Initial trajectories for (a) bottom layer and (b) middle layer. Green lines visualise the printing head path, and brown dots are checkpoints

Each layer may have a different number of points or vary in width and length (Fig. 5.4). Therefore, when moving from the bottom layer to the upper layer, increasing the Z axis is not feasible, as this could lead to printing misalignment. To avoid any collisions, our trajectory always

begins from the top-left position, as seen from the camera's perspective, and proceeds to the bottom-right.

Our method entails creating a zig-zag trajectory that is either vertical or horizontal. In our study, we set the spacing coefficient of the zig-zag pattern filling to 3 pixels. To create a printing path, our algorithm arranges the vertices of each layer with a given spacing step along the X or Y axis. Next, it decides which way to go for each line, going left or right (or top/bottom) alternately. We build trajectories for every layer in a single run to accommodate for any motions that may occur during layer transitions. The initial trajectory is only modified when necessary and is never reproduced after it has been formed.

We create checkpoints and append them to a num array when we generate the printing path. The Python library NumPy [79] provides vectorised linear algebra operations. With NumPy, we can update the matrix of the trajectories without iterating over it. The shortest distance that the printer can travel while extruding the selected material determines at which location the checkpoints should be placed.

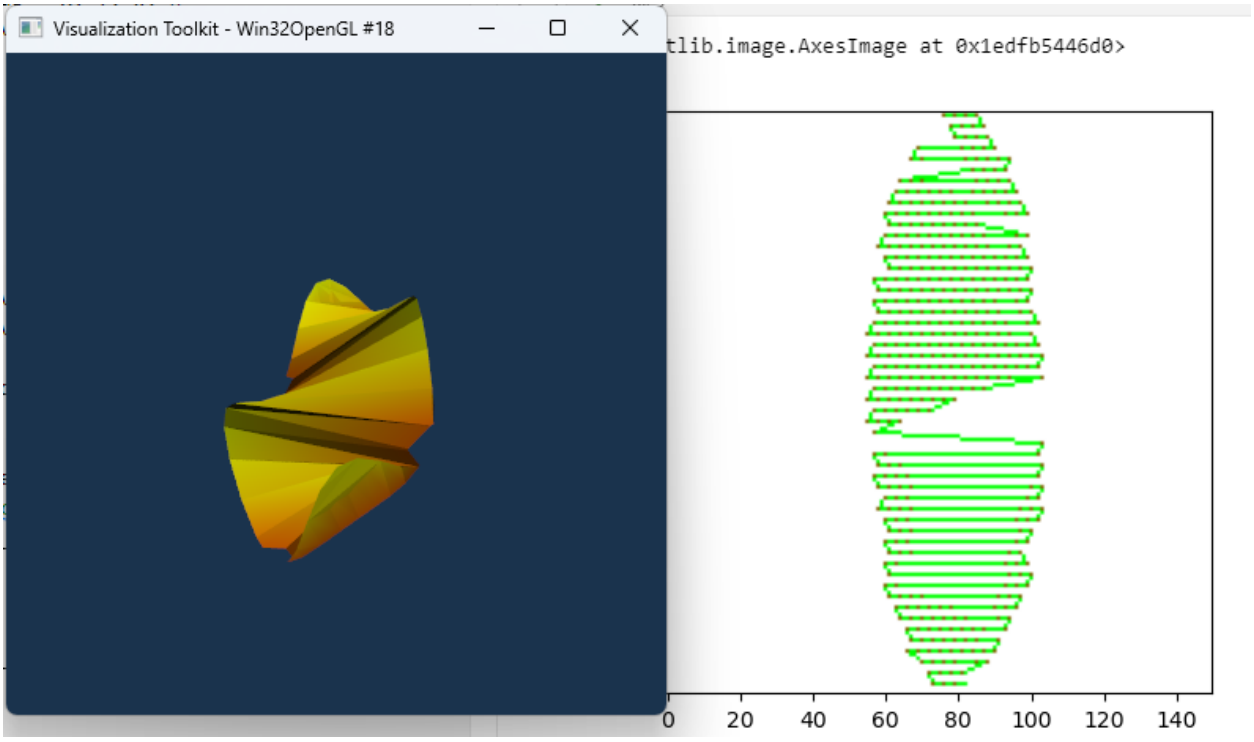


Figure 5.5 Trajectory for more complex surfaces. In the left picture – the side view on one of the top layers rotated towards Z-axes. On the right – the trajectory viewed from the top bypasses all the bumps, ensuring the same level of filling

In Figure 5.5, we provide an example of a complex and curved surface. In the case when one side of the wound has higher bumps, our algorithm avoids it by either moving over it or getting around if that area is too high.

To ensure that all our points are inside of the wound, we represent each layer as a 2D contour, and using the OpenCV method pointPolygonTest(), we verify whether each point is inside the contour.

5.3 Optimal Path Planning

This section covers the selection process of optimal path design. Below, we reviewed how different zig-zag orientations may decrease the total printing process. For our experiments, we used four real wound contours, which we showed in Chapter 3: Figures 3.10 and 3.12. The used contours are in 2D format only; because of this, we also analyse and validate one-layered trajectories. A printing path is an ordered set of XY points. To simulate checkpoint localisation, we inserted them in every second point on the path.

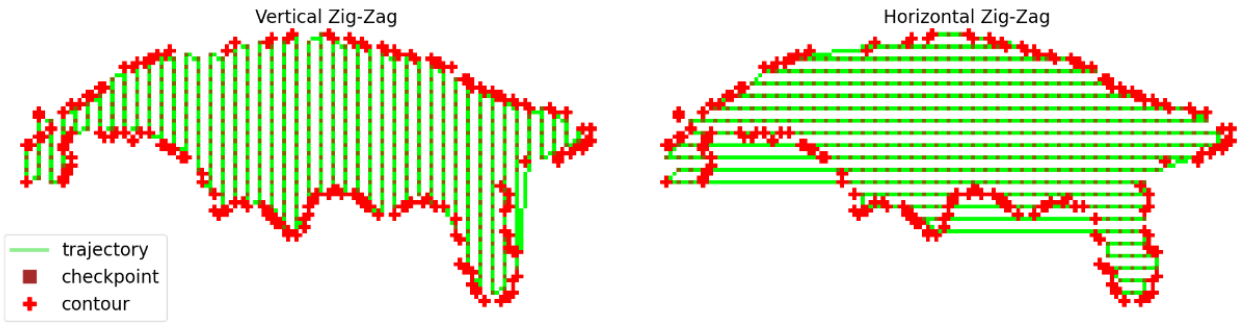


Figure 5.6 Two examples of trajectory: vertical and horizontal

In Figure 5.6, we visualise two examples of trajectory orientation. As a baseline, we used the biggest wound area from the list, with a total area of 3995.5 px (calculated with OpenCV *contourArea()*).

To find the best trajectory, we generate two paths in horizontal and vertical orientations and evaluate them by summing up Euclidean distances between two points in the generated length. The one with the shortest travelling path is the most optimal printing trajectory. Considering that we calculate and evaluate printing plans on the fly, it's reasonable to measure the computational power

required for calculations (Fig. 5.7). From Figure 5.7, the processing time depends on the overall distance but does not exceed 3 ms for two orientations, even for the biggest wound.

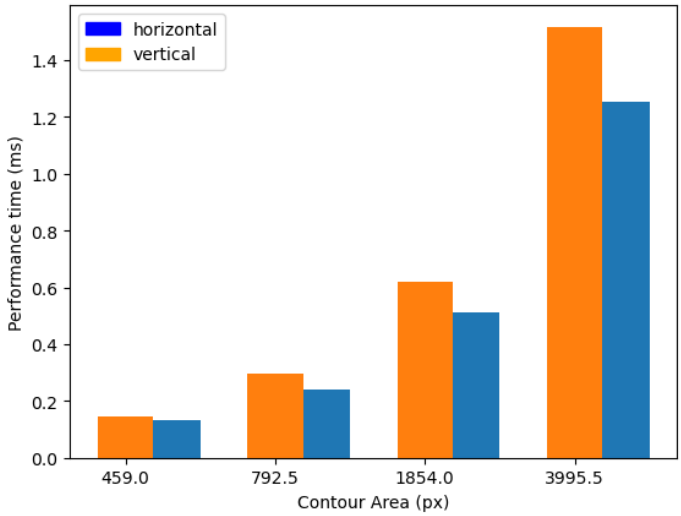


Figure 5.7 The time needed to calculate two printing path orientations

Additionally, we reviewed how the number of checkpoints depends on the printing arrangement (Fig. 5.8a) and estimated the total length per orientation for different contours (Fig. 5.8b). From the figure, we see the more notable difference between less straightforward and round wounds. It is through that the checkpoints are always inside of the wound, pointing on the part of the filament extrusion way, but the printhead may move without printing, too.

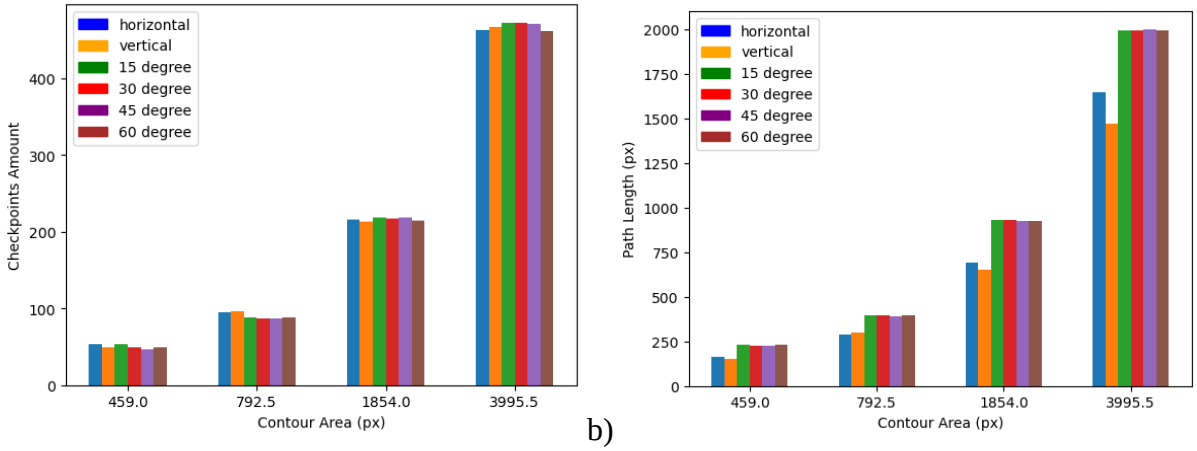


Figure 5.8 Checkpoints amount has linear dependency from the total area and is almost unaffected by the selected pattern (a). However, we observed that path length varies based on the pattern orientation (b)

In conclusion to this section, the optimal path is selected by determining the shortest path using the Euclidean distance metric. The fast processing allows for iterating through the options of different hatching patterns.

5.4 Trajectory Updating

The initially generated trajectory is updated in a separate processing thread (thread #4). The input for this updating thread consists of the original trajectory and movement coefficients (Fig. 5.9). The output is an updated trajectory that replaces the original one in further iterations. If no movement occurs and the trajectory remains unchanged, the thread triggers a flag indicating that the environmental conditions have not changed. As a result, the following thread, responsible for robot control, will continue to use the existing trajectory. If an error occurs, the system will handle it accordingly.

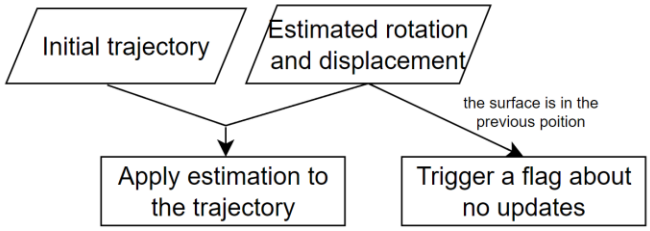


Figure 5.9 Diagram of the trajectory updating process

To rotate the trajectory in 3D space, a rotation matrix is defined for each axis. The rotation sequence defines individual rotation matrices for each axis of rotation (X, Y, and Z) (Equation 5.1-5.3). A specified rotation point is considered when applying the rotation (Equation 5.4, 5.6).

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \tag{5.1}$$

$$R_y = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{5.2}$$

$$R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Where θ is a rotation angle in radians.

$$A_T = A - P_R \tag{5.4}$$

$$A_R = A_T \cdot R \tag{5.5}$$

$$A_U = A_R + P_R \tag{5.6}$$

Where A is a 3D matrix of the initial trajectory, P_R is a 3D vector of rotation point, A_T– is a translated trajectory in 3D A_R, – is a rotated trajectory, and A_U – is an updated trajectory. R – is a rotation matrix for the axis at which rotation appeared: use R_x if roll rotation happened, R_y – if pitch rotation, and R_z – for yaw rotation.

To change the position of the trajectory, we repeat a similar operation over an updated 3D trajectory (Equation 5.7).

$$A_{U'} = A_U + P_T \tag{5.7}$$

Where P_T is a 3D vector of the transition point.

All the vector and matrix operations above were vectorized with the use of NumPy, which handles multiple entities of pixels simultaneously.

In Figure 5.10, we demonstrate two examples of how the coordinates of the initial trajectory (Fig. 5.6a) changed when the surface relocated (a) or rotated (b). We can observe that the shape of the trajectory is the same, and the only thing that changed is its position relative to the previous boundaries.

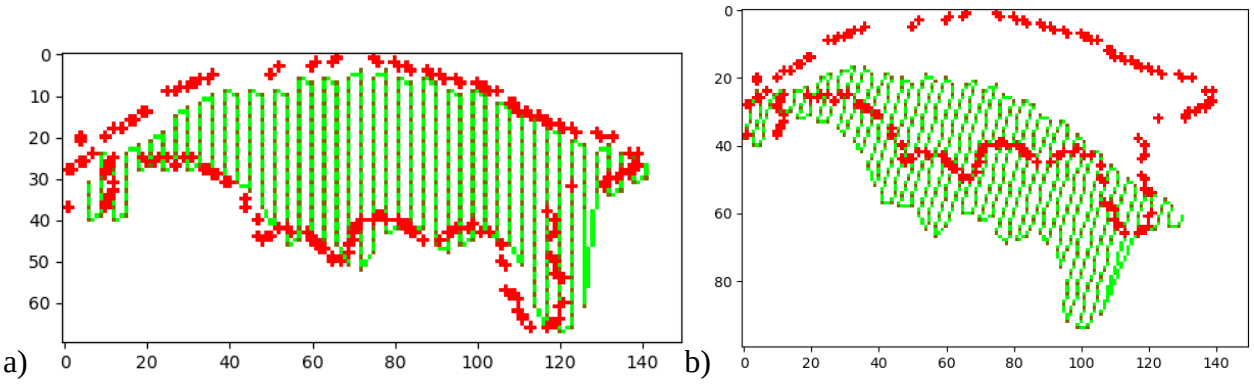


Figure 5.10 2D examples of the initial trajectory displacement due to the surface: (a) movement to the right and down (X+Y+), (b) rotation at 15 degrees at the Z-axis. Red crosses represent the initial contour placement on the scene

To provide a more comprehensive view of the rotation example, we added depth information to the visualisations. In Figure 5.11, we rotated the trajectory from Figure 5.4 (b) at the Y-axis. This slice has the same depth value for every path point. But when we added a Y rotation, the top part of the printing plan went up, and the bottom part went down. We can see it with the darker colour of the upper part, which slowly lights up towards the lowest point.

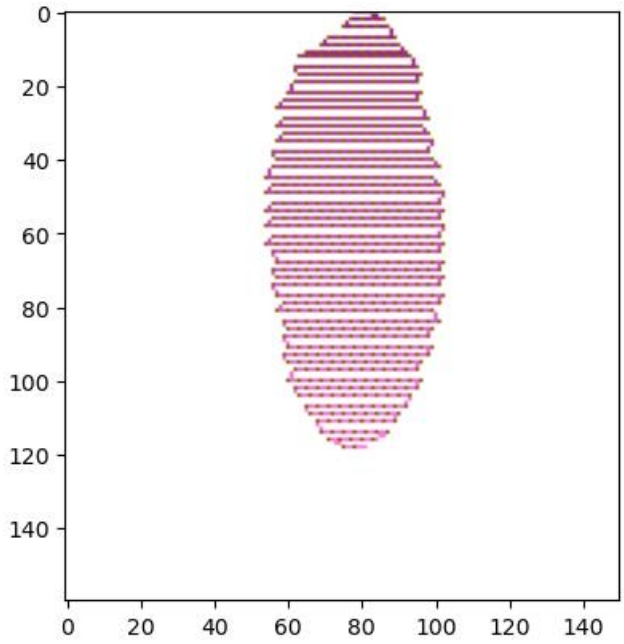


Figure 5.11 3D example of 40 degrees pitch rotation. The darker colour illustrates the upper position over the other side of the trajectory. There is an illusion of shrinkage, as the original trajectory (Fig. 5.4b) is 150 px in length, and the updated shape is less than 120 px in length

5.5 Conclusions

The primary goal is to design the most optimal 3D printing path for biomaterials. The nozzle of a 3D bioprinter navigates a wound that is modelled as a layered object, representing the hypodermis, dermis, and epidermis, the three main layers of skin. We selected a zig-zag pattern for our demonstrations as it is simple to reproduce and operate.

We introduced a checkpoint system as a progress status recorder. With the checkpoints, the system can follow the unfinished printing parts. Additionally, each checkpoint contains all the necessary information for printing commands, including speed, materials, etc., providing great flexibility on desired adjustments.

Optimal path planning means evaluating different zig-zag orientations to minimise the total printing time. We used the Euclidean distances between points in the trajectory to select the shortest path. The trajectory is continuously updated in a separate processing thread, which recalculates the trajectory based on movement coefficients and replaces the original trajectory as needed.

The generation of an initial trajectory from a 3D mesh, real-time adjustments via a checkpoint system, and ongoing trajectory updates are all vital components in the development of an ideal 3D printing path. This system adjusts to changes in the surface of the wound, optimises the printing path, and manages computational resources effectively to ensure accuracy and speed printing.

6 6-DOF Robotic Arm Control

Summary: This chapter provides an overview of an adjustable control system designed for a 6-DOF robotic arm, specifically the SEED S6H4D robot. It emphasises the difficulties in controlling non-stationary printing trajectories and highlights the task of adjusting for variations in orientation, speed, and direction. The chapter also goes over the robot's communication protocol, illustrating the need for proper data conversion and command structure. Finally, it discusses safety protocols, the robot's operational limitations, and the integration of all the discussed modules into the responsive operating algorithm to cope with dynamic printing tasks.

6.1 Introduction

This chapter aims to detail an adjustable control system for a 6-DOF robotic arm by the example of controlling a SEED S6H4D robot (Fig. 6.1). The main challenge of this objective is to manipulate the robot arm, taking into consideration a non-stationary printing trajectory. Traditional 3D printers manoeuvre a predefined printing trajectory, which is relatively straightforward since the printer's movements are restricted to a fixed plane or workbench. However, when working with a 6-DOF robotic arm, the freedom of movement, as well as the complexity of the command lines, increases. Moreover, the non-stationary printing trajectories mean that the arm must be able to adapt and handle changes in direction, speed, and orientation to react instantly, within a 33 ms timeframe (before a new frame is captured), but move smoothly. This is incredibly challenging when printing complex geometries or when the printing surface itself is not fixed and moves unpredictably.

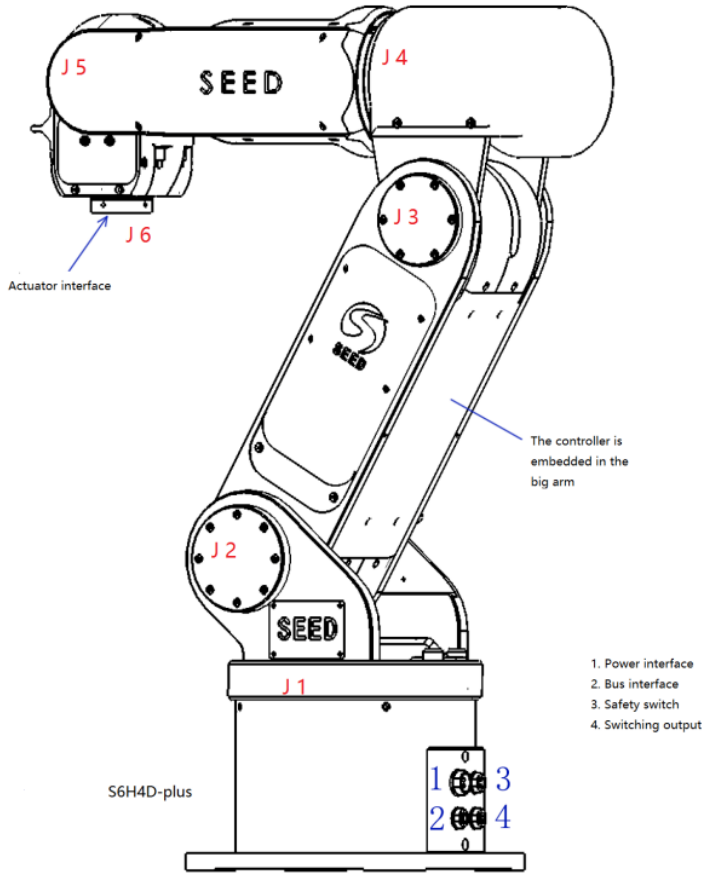


Figure 6.1 A SEED S6H4D robot arm **[80]**

Considering the complexity of operating the 6-DOF robotic arm, we should mention that such a mechanism is a moving machine with a certain strength and speed. The SEED robot appears with no sensors to avoid hitting its surroundings. In dynamic and random environments, serious injuries are possible, and we handle them with specific emergency protocols.

6.2 Communication Protocol

The control and drive system are embedded in the body of the SEED S6H4D robotic arm. The robot's system is controlled via a host computer's USB using a predefined communication protocol. The communication uses a baud rate of 115200bps. No additional software was used. The robot uses a serial communication interface for the input control instructions. The robot's responses are a single-byte string sent in a random manner. This is a keynote to mention because, with our

robot, we cannot receive its coordinates as we do with other robot arms. We calculate its current position based on the home coordinates plus the commanded displacements over the process.

Geometry code (G-code) is a programming language for 3D printing and other automated machines. As with other robots, the SEED robot uses G-codes. The G-codes we use for our project are:

- G1 – linear interpolation or move in a straight line. The parameters are:
    - X, Y, Z – cartesian coordinates (mm).
    - A, B, C – J4, J5, J6 (degrees).
    - F – speed (mm/min).
- G4 – pause all the processes:
    - P – pause time (ms).
- G54 – set workbench coordinates.
    - X, Y, Z – set these coordinate values as (0, 0, 0).

The one serial communication instruction has a fixed length of 48 bytes. The total memory buffer is 9600 bytes or 200 command lines. In-buffer commands run in a first-in, first-out (FIFO) manner. If the buffer exceeds its capacity, new commands will not be processed until there is space available. Each instruction includes all necessary control and motion functions of the robot arm. Certain data, coordinates and other parameters from the list above are represented by floating point numbers, each occupying 4 bytes within the 48-byte instruction. We will provide a data conversion function below.

While the robot accepts human-readable G-codes, the commands sent directly from the host computer via serial port should be encoded to bytes with the proper frame's header and tail (Table 6.1). The instructions sent by the host computer to the robotic arm are divided into two categories: one is a motion instruction, and the other one is a system setup.

Corresponding G-code	Frame header (1 byte)	Instruction codes (2 bytes)	Acceptable parameters (max 40 bytes / 10 parameters)	Frame tail (1 byte)
G1	238	1*, 1	X, Y, Z, A, B, C, F	239
G4	238	6*, 0	P	239

G54	252	20, 1	X, Y, Z	253

* is represented by a character value.

Table 6.1 Proper command structure

To handle this structure in Python, we first create a list with forty-eight zeroes. Every zero in this step is a placeholder for 1 byte of data. Every item in the list has its own index. We use these indexes to replace 0's with the desired bytes.

Index 0 represents the header, and index -1, or 47, is the frame tail. From Table 6.1, we can notice that motion commands start with a "238" value and end with "239", but system setup commands begin with "252", ending with "253".

With the specified instruction codes, the robot system can determine the expected movement pattern. Instruction codes take 2 bytes of data or index 1 and 2 in our list. For our list of G-codes, the code values are constant.

The output from Chapter 5 – trajectory coordinates are float numbers. Every parameter value in the end is a hexadecimal string prefixed with "0x". To convert float to a hexadecimal string, we complete 3 steps:

1. Format float numbers into the IEEE 754 binary32 single-precision format with a standard size of 4 bytes (or 4 * 8 = 32 bits), where the first 1 bit is a signed value (0 – positive, 1 – negative), next 8 bits are devoted to a biased exponent (to keep stored exponent), the rest 23 bits – a mantissa.
2. From the previous step, we receive a byte array with a length of 4. Now, every byte is converted to a lowercase hexadecimal string prefixed with "0x".
3. In the last step, we convert the hexadecimal representation to an integer with 16 bases. This is necessary to convert resulted values to a bytes string.

After conversion is done, we place every byte from the result above within the specified index: X or P parameter values are placed inside the range of 3-6 indexes, Y takes 7-10 indexes, Z is 11-14, A is 15-18, B is 19-22, C is 23-26, and F is 43-46.

To create the final byte string to send to the robot, we divide our list into three sections: the first symbol, the instruction code character, and the rest of the frame. This is required because

characters can't be converted to byte strings as numbers. Characters should be encoded with UTF-8 encoding.

The following G-code, after the formatting algorithm, would be like this:

G1 X200 Y-50 Z180 A0 B10 C20 F2000 →
b'\xee1\x01\x00\x00HC\x00\x00H\xc2\x00\x004C\x00\x00\x00\x00\x00\x00\x00\x00\xa0A\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\xfaD\xef'

To make the robot arm respond to every updated command and avoid moving to the unactual coordinates, we send one command at once (one command is one checkpoint). To be sure that the robot will respond only to the latest command, we clear the buffer before sending the new command. This helps to circumvent queues.

6.3 Robot Operating System

To operate the robot arm, we should know its capabilities and limitations:

- The SEED S6H4D robot arm's maximum linear speed is up to 60,000 mm/min, J1-J4: 120 degrees/s, and J5-J6: 240 degrees/s. For our experiments, we did not raise the speed higher than 3500 mm/min.
- To determine the range limit for surface movements, we should know joint ranges. J1: -150°-150°; J2: -65°-110°; J3 167°-167°; J4 27°-188°; J5 -270°-270°; J6 -34°-208°.

Before the start of the operation, the system sets a new workbench coordinate to the approximate centre of the wound based on the segmentation mask (output from Chapter 3).

We discussed how to create and send commands for the robot, but in this section, we will explain how to use the formatting function in the natural environment. A robot arm movement function runs in a separate thread (5th in total), but it waits until the trajectory updating function returns its first output since the robot's move commands can't be created without the actual coordinates. To remind the Chapter 5 output format: the function returns a list of the world coordinates, together with current angles of the surface ([(x_1, y_1, z_1, α_1, β_1, γ_1), …, (x_n, y_n, z_n, α_n, β_n, γ_n)]; and there other two codes: 1 – printing path is the same and None – pointing to appeared errors.

The inside of the moving function (Fig. 6.2) includes a loop, which goes through all the coordinates from the list and, in the end, formats instructions and writes to the robot. The main components in the loop are a flag receiver and a coordinate index. The flag receiver constantly monitors the trajectory updating thread: in case of the error code, the robot does not continue to move and keeps the last index (checkpoint) in the memory; if the receiver gets an updated version of the coordinates, the robot will start moving according to the new coordinates in the next loop iteration; same, if it gets a 1 code, meaning the trajectory is the same, so the robot will continue using the previous coordinates. The loop ends after the index reaches the last checkpoint from the list.

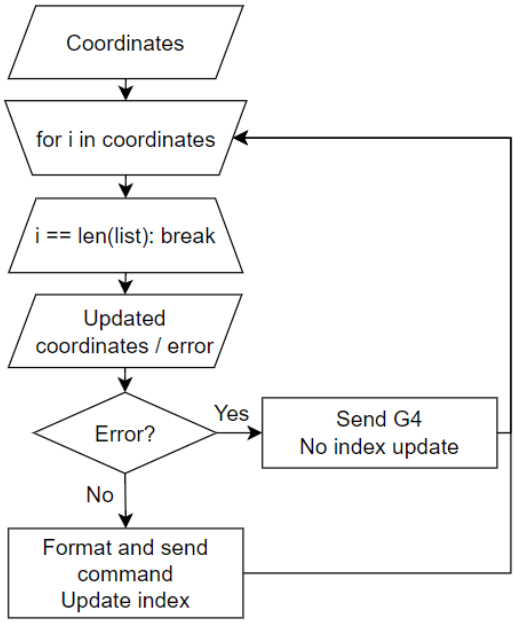


Figure 6.2 Diagram of the move robot arm function to go through the list of coordinates

After the work is completed, we set a workbench/home coordinate to X202.5 Y225 Z113 A0 B0 C0.

We use a pause command to avoid the robot's unpredictable behaviour due to incomplete/impossible commands. The pause timeline equals 1 ms, as it is a minimal processing time for the trajectory updating function. If the error is not resolved during this timeframe, the waiting time will be repeated.

Additionally, we utilise breaks when the surface moves too fast. The current threshold for the positioning is 5 cm/s and 40° for rotations.

6.4 Integrated System

To conclude all our work, we integrated all the objectives into one script. The experimental system for the research is presented in Figure 6.3.

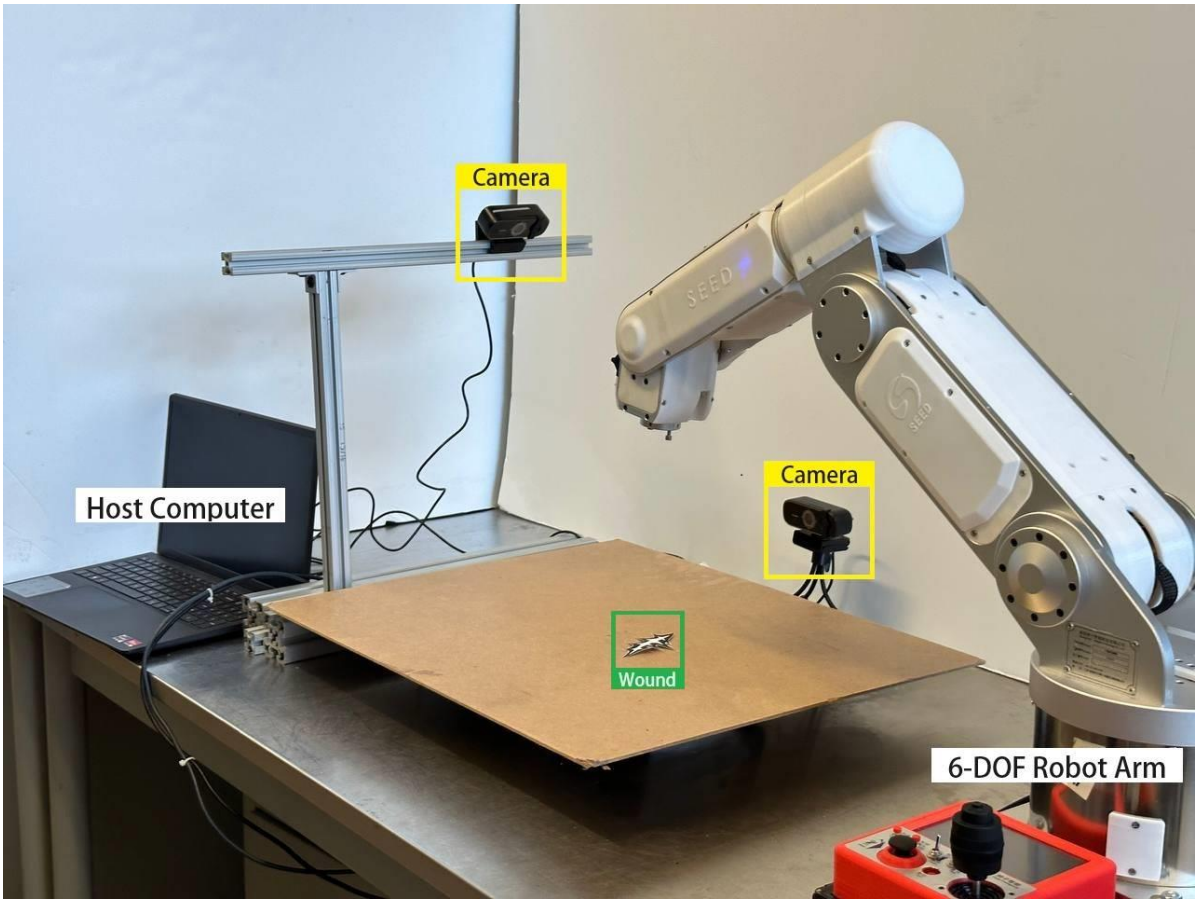


Figure 6.3 Experimental setup: 2 cameras, a simulated wound, a host computer with the running script, connected to the 6-DOF SEED robot arm

The experimental results are presented as frames from the live video of the moving in 5-DOF simulated wound and the robot response to those movements in Figure 6.4 and Figure 6.5. The performed movements include yaw and roll rotations (Fig. 6.4) and displacements in X-, Y-,

and Z-axes (Fig. 6.5). The target object is a naturally moved hand. Thus, no measurement mechanisms were present to validate the precision of the robot positioning.

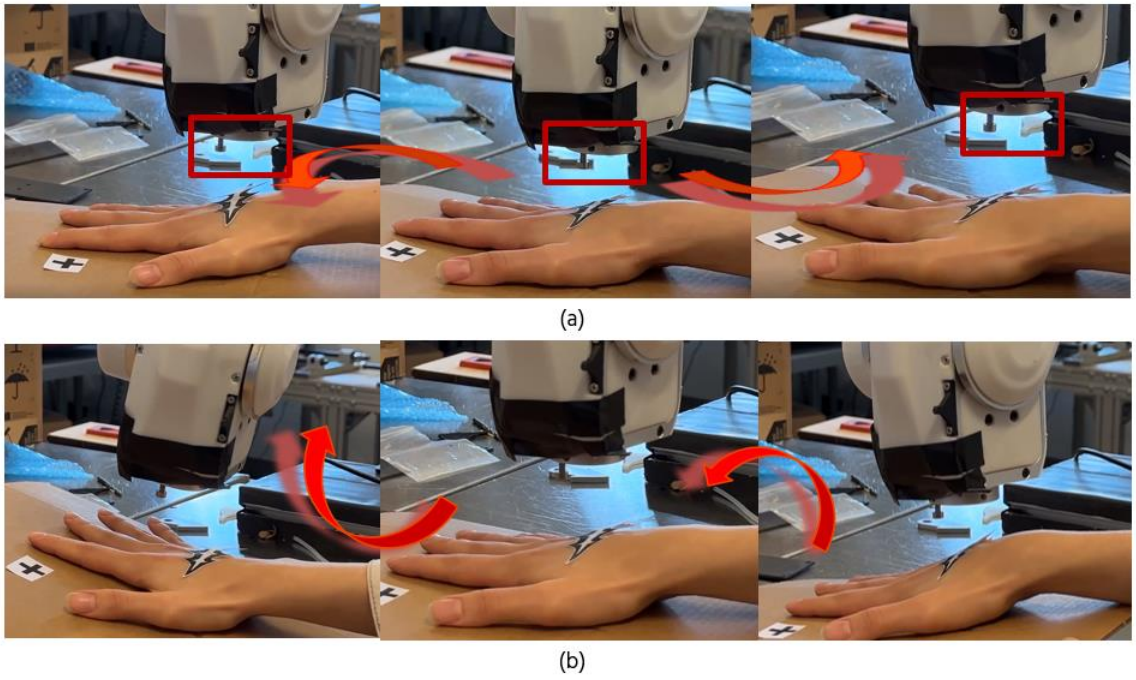


Figure 6.4 Yaw (a) and roll (b) rotations of the hand. The robot follows the rotation. To notice yaw rotation, focus on the position of the bolts and holes at the J6.

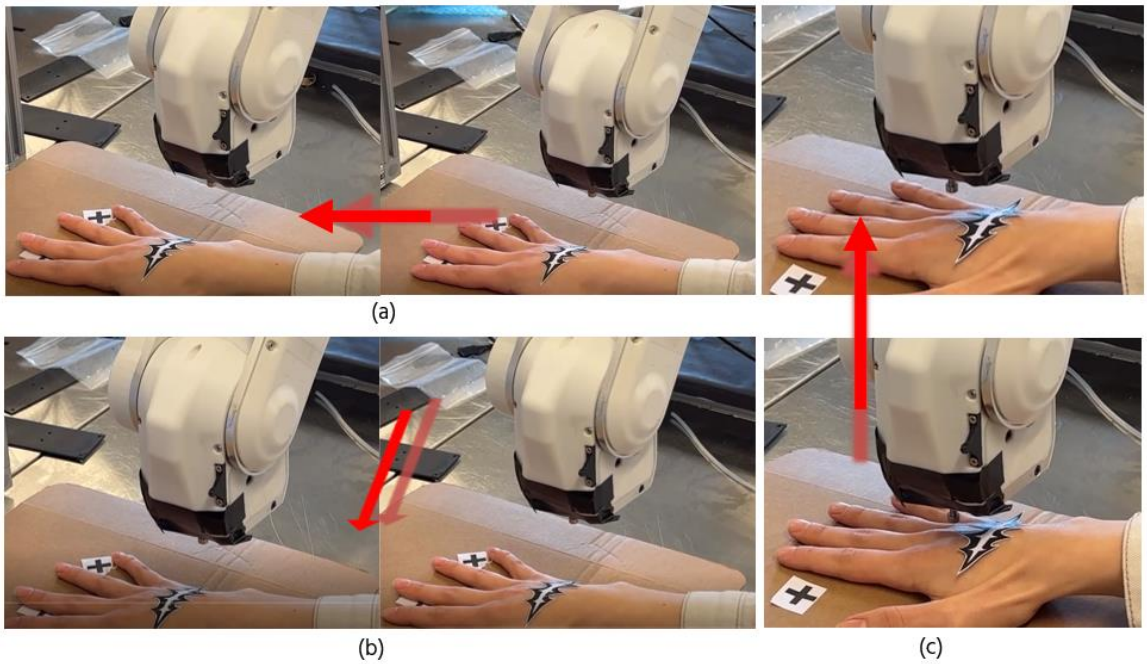


Figure 6.5 X-axis (a), Y-axis (b), and Z-axis (c) movements. The left image is the initial hand and robot position for X and Y movements (a,b). For Z movement (c), the bottom is the initial position, and the top is the final

6.5 Conclusions

In this Chapter, we looked at the challenges of managing non-stable printing trajectories with a 6-DOF robotic arm using the SEED S6H4D robot arm as an example. The robot's requirement for rapid direction and orientation adaptation presents a hurdle.

We went into detail about the SEED S6H4D's communication protocol, which is based on a serial communication interface and special command instructions. For these instructions, we stressed the significance of appropriate data conversion in accordance with the documentation. G1 is used for linear movement in our studies, G4 is used to pause operations, and G54 is used to set workbench coordinates.

To guarantee safe and sensible operation, the robot's capabilities, such as its joint ranges and speed limits, were taken into account. We emphasised a workbench coordinate initialisation procedure that uses a segmentation mask – the output from Chapter 3. The movement function runs in a multi-threaded environment to guarantee that the robot reacts quickly to updated orders.

All things considered, the control system designed for the SEED S6H4D robotic arm shows a strong method for handling demanding and dynamic printing jobs. The robotic arm can perform the simulation of printing work because of the combination of a responsive operating algorithm and a formatting function for the communication protocol. The versatility of this technology creates new opportunities for complex 3D printing applications and advanced AM in addition to in-situ bioprinting, opening the door for more advancements in robotic control systems.

7 Conclusion and Future Work

7.1 Statement of the Problem

Our research addresses the difficulties in direct 3D bioprinting on human bodies, which move frequently and unpredictably. The main challenge is to precisely apply biomaterials without creating flaws in the printed tissues. Since traditional 3D bioprinting technologies are made for static surfaces, it takes creative tracking systems and control algorithms to adapt them to dynamic, non-planar surfaces. To guarantee the effectiveness and quality of the printed tissues, these systems need to precisely align the bio-ink nozzle with the moving target surface. The foundation for creating an advanced tracking system that can manage the difficulties of bioprinting under such demanding circumstances is laid out in our problem statement.

This work sought to find an applied solution to the problem and explore time-effective methods to perform real-time analysis.

7.2 General Conclusions

i. The models with U-Net architecture are very reliable and fast. It is a good option for an application that requires instantaneous responses. The drawback of the medical imaging Deep Learning model is that it often lacks data on which to be trained. A combination of Semi-Supervised Learning methodology and data augmentation can improve the model training experience.

ii. 3D space stores information about objects' width, height, and depth. Regular cameras are capable of capturing 2D width and height only. The third dimension builds up with the multiple-cameras system and is calculated from the triangular relations between cameras. With the obtained 3D observation, we can estimate 3D movements from 2D images. Our experiments showed high precision for tracking chaotic 6-DOF movements in the simulation. However, in the natural environment, the system should be closely monitored and regularly calibrated to maintain high accuracy.

iii. Optimal trajectory generation and its adjustments are possible with the proper data representation. Instead of processing dense point clouds, we reproject it as an Nx3 matrix of coordinates. We divide down the problem and created a stack of 2D slices from the original matrix. Each layer is treated individually, providing flexibility to arrange the

printing path by a unique rule. Each point on the route is represented as an N-dimensional vector of 3D coordinates and printing material characterisation. With such a data representation, 3D printing becomes a fully controllable but automatic process.

iv. Communication with the robot is implemented using a specific protocol: messaging. The combination of operating algorithms and a command formatting function allows the robotic arm to respond to immediate commands instead of following predefined instructions.

7.3 Thesis Contributions

This thesis suggests an applied system for the additive manufacturing field to perform various tasks in an active condition. This work introduces a sophisticated motion-tracking system designed for 6-DOF robot arms. Below, the objective-based contribution made during the program:

a. Wound Area Detection: For objective I, developed an advanced DL model for medical imaging's real-time wound segmentation. With a Dice coefficient of 0.765 and a processing time of 9 ms per image, this U-Net-based model outperformed pure CV techniques in terms of accuracy and speed.
b. 6-DOF Motion Detection and Estimation: In objective II, a method for detecting and estimating motion in a 6-DOF setup was created using SIFT combined with FLANN algorithms. This system accurately tracks complex 3D movements and works well in both simulated and real environments. Furthermore, the system's 70 ms processing time guarantees effective operation, fitting it for real-world robotics applications.
c. Printing Trajectory Generation: Designed a novel 3D printing path optimisation and monitoring solution for objective III. Real-time trajectory adjustments and a dynamic checkpoint system that enables last-minute changes during printing are incorporated into this system. In addition to monitoring progress, the checkpoint system stores detailed information regarding printing commands, such as material type, speed, and other important variables, providing flexibility for any necessary modifications.
d. 6-DOF Robotic Arm Control: Objective IV brings an engineered control system for a 6-DOF robotic arm tailored for dynamic 3D printing applications. By leveraging a multi-threaded communication protocol and implementing movement adjustments predicated on instantaneous segmentation information, the system offers an efficient framework for

managing intricate, non-stable printing paths. This opens up new opportunities for in-situ bioprinting and advanced manufacturing, enabling accurate, flexible, and effective robotic control in dynamic situations.

7.3.1 Publications

A. Czekanski and K. Zakharova, “3D Bioprinting on Moving Surfaces Using Computer Vision,” in Canadian Society for Mechanical Engineering International Congress, 6, 2023.

A. Czekanski and K. Zakharova, “Wound Segmentation With Semi-Supervised U-Net Model For Unseen Perspective Tracking,” in Canadian Society for Mechanical Engineering International Congress, (*in press*) 2024.

A. Czekanski and K. Zakharova, “Algorithm For Fully Automatic In-Situ 3D Bioprinting,” in Canadian Society for Mechanical Engineering International Congress, (*in press*) 2024.

7.4 Future Work

There are several potential suggestions for future work for the tracking system for 3D bioprinting on moving surfaces application:

- Investigate more advanced CV tracking algorithms that might be more time-consuming but highly accurate. Overview current state-of-the-art DL algorithms and propose a way to convert them to online models.
- Try out a hardware acceleration that could reduce the processing time for the real-time application.
- Review 3D modelling solutions for concurrent wound shape reconstruction. While printing on soft tissue, it is necessary to maintain the actual form of the printing area, which might be changed under different circumstances.
- Experiment with the 6-DOF vibration platforms, which can provide validation data for the real-world conditions system.

References

[1] M. A. Shampo and R. A. Kyle, ‘Joseph E. Murray—Nobel Prize for Organ Transplantation’, *Mayo Clin. Proc.*, vol. 76, no. 3, p. 240, Mar. 2001, doi: 10.4065/76.3.240.

[2] C. Mandrycky, Z. Wang, K. Kim, and D.-H. Kim, ‘3D bioprinting for engineering complex tissues’, *Biotechnol. Adv.*, vol. 34, no. 4, pp. 422–434, Jul. 2016, doi: 10.1016/j.biotechadv.2015.12.011.

[3] I. T. Ozbolat and M. Hospodiuk, ‘Current advances and future perspectives in extrusion-based bioprinting’, *Biomaterials*, vol. 76, pp. 321–343, Jan. 2016, doi: 10.1016/j.biomaterials.2015.10.076.

[4] A. Harding, A. Pramanik, A. K. Basak, C. Prakash, and S. Shankar, ‘Application of additive manufacturing in the biomedical field- A review’, *Ann. 3D Print. Med.*, vol. 10, p. 100110, May 2023, doi: 10.1016/j.stlm.2023.100110.

[5] Z. Xie, M. Gao, A. O. Lobo, and T. J. Webster, ‘3D Bioprinting in Tissue Engineering for Medical Applications: The Classic and the Hybrid’, *Polymers*, vol. 12, no. 8, p. 1717, Jul. 2020, doi: 10.3390/polym12081717.

[6] C. Dikyol, M. Altunbek, P. Bartolo, and B. Koc, ‘Multimaterial bioprinting approaches and their implementations for vascular and vascularized tissues’, *Bioprinting*, vol. 24, p. e00159, Dec. 2021, doi: 10.1016/j.bprint.2021.e00159.

[7] M. Schouten, G. Wolterink, A. Dijkshoorn, D. Kosmas, S. Stramigioli, and G. Krijnen, ‘A Review of Extrusion-Based 3D Printing for the Fabrication of Electro-and Biomechanical Sensors’, *IEEE Sens. J.*, vol. PP, Dec. 2020, doi: 10.1109/JSEN.2020.3042436.

[8] S. Park, S. Kim, and J. Choi, ‘Development of a multi-nozzle bioprinting system for 3D tissue structure fabrication’, in *2015 15th International Conference on Control, Automation and Systems (ICCAS)*, Oct. 2015, pp. 1874–1877. doi: 10.1109/ICCAS.2015.7364668.

[9] M. Ali, N. Mir- Nasiri, and M. Ko, ‘Multi-nozzle extrusion system for 3D printer and its control mechanism’, *Int. J. Adv. Manuf. Technol.*, vol. 86, Sep. 2016, doi: 10.1007/s00170-015-8205-9.

[10] M. Albanna *et al.*, ‘In Situ Bioprinting of Autologous Skin Cells Accelerates Wound Healing of Extensive Excisional Full-Thickness Wounds’, *Sci. Rep.*, vol. 9, no. 1, p. 1856, Feb. 2019, doi: 10.1038/s41598-018-38366-w.

[11] M. Xie *et al.*, ‘In situ 3D bioprinting with bioconcrete bioink’, *Nat. Commun.*, vol. 13, no. 1, p. 3597, Jun. 2022, doi: 10.1038/s41467-022-30997-y.

[12] Z. Zhu *et al.*, ‘3D Printed Functional and Biological Materials on Moving Freeform Surfaces’, *Adv. Mater.*, vol. 30, p. 1707495, Apr. 2018, doi: 10.1002/adma.201707495.

[13] A. Shembekar, Y. J. Yoon, A. Kanyuck, and S. Gupta, ‘Generating Robot Trajectories for Conformal 3D Printing Using Non-Planar Layers’, *J. Comput. Inf. Sci. Eng.*, vol. 19, Mar. 2019, doi: 10.1115/1.4043013.

[14] R. C. Luo and P.-K. Tseng, ‘Trajectory generation and planning for simultaneous 3D printing of multiple objects’, in *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE)*, Jun. 2017, pp. 1147–1152. doi: 10.1109/ISIE.2017.8001407.

[15] K.-C. Ying, P. Pourhejazy, C.-Y. Cheng, and R.-S. Syu, ‘Supply chain-oriented permutation flowshop scheduling considering flexible assembly and setup times’, *Int. J. Prod. Res.*, vol. 61, no. 1, pp. 258–281, Jan. 2023, doi: 10.1080/00207543.2020.1842938.

[16] K.-C. Ying, F. Fruggiero, P. Pourhejazy, and B.-Y. Lee, ‘Adjusted Iterated Greedy for the optimization of additive manufacturing scheduling problems’, *Expert Syst. Appl.*, vol. 198, p. 116908, Jul. 2022, doi: 10.1016/j.eswa.2022.116908.

[17] N. Ganganath, C.-T. Cheng, K.-Y. Fok, and C. K. Tse, ‘Trajectory planning for 3D printing: A revisit to traveling salesman problem’, in *2016 2nd International Conference on Control, Automation and Robotics (ICCAR)*, Apr. 2016, pp. 287–290. doi: 10.1109/ICCAR.2016.7486742.

[18] A. Munteanu *et al.*, ‘A Study on the Errors of 2D Circular Trajectories Generated on a 3D Printer’, *Appl. Sci.*, vol. 11, no. 24, Art. no. 24, Jan. 2021, doi: 10.3390/app112411695.

[19] X. Lu *et al.*, ‘Reconstruction method and optimum range of camera-shooting angle for 3D plant modeling using a multi-camera photography system’, *Plant Methods*, vol. 16, no. 1, p. 118, Aug. 2020, doi: 10.1186/s13007-020-00658-6.

[20] J. Yoon, C. Koch, and T. Ellis, ‘ShadowFlash: an approach for shadow removal in an active illumination environment.’, Jan. 2002.

[21] ‘Lighting for background removal’. Accessed: Jun. 02, 2024. [Online]. Available: https://www.36pix.com/lighting-tips-for-background-removal/

[22] A. Bielawny, ‘Reflectors in lighting design: Reflector-based non-imaging optics for lighting applications’, *Adv. Opt. Technol.*, vol. 8, no. 6, pp. 469–481, Dec. 2019, doi: 10.1515/aot-2018-0052.

[23] K. Yoon, J. Seol, and K. G. Kim, ‘Removal of Specular Reflection Using Angle Adjustment of Linear Polarized Filter in Medical Imaging Diagnosis’, *Diagnostics*, vol. 12, no. 4, p. 863, Mar. 2022, doi: 10.3390/diagnostics12040863.

[24] S. Lee, K. Yoon, J. Kim, and K. G. Kim, ‘Specular Reflection Suppression through the Adjustment of Linear Polarization for Tumor Diagnosis Using Fluorescein Sodium’, *Sensors*, vol. 22, no. 17, p. 6651, Sep. 2022, doi: 10.3390/s22176651.

[25] C. Lu and M. Drew, ‘Shadow Segmentation and Shadow-Free Chromaticity via Markov Random Fields’, *Color Imaging Conf.*, vol. 13, Jan. 2005, doi: 10.2352/CIC.2005.13.1.art00024.

[26] G. Finlayson and S. Hordley, ‘Color constancy at a pixel’, *J. Opt. Soc. Am. A Opt. Image Sci. Vis.*, vol. 18, pp. 253–64, Mar. 2001, doi: 10.1364/JOSAA.18.000253.

[27] J. Demantké, C. Mallet, N. David, and B. Vallet, ‘DIMENSIONALITY BASED SCALE SELECTION IN 3D LIDAR POINT CLOUDS’, *Int. Arch. Photogramm. Remote Sens. Spat. Inf. Sci.*, vol. XXXVIII-5-W12, pp. 97–102, Sep. 2012, doi: 10.5194/isprsarchives-XXXVIII-5-W12-97-2011.

[28] A. Al-Rawabdeh, F. He, and A. Habib, ‘Automated Feature-Based Down-Sampling Approaches for Fine Registration of Irregular Point Clouds’, *Remote Sens.*, vol. 12, no. 7, Art. no. 7, Jan. 2020, doi: 10.3390/rs12071224.

[29] Lu-Xingchang and Liu-Xianlin, ‘Reconstruction of 3D Model Based on Laser Scanning’, Jan. 2006, pp. 317–332. doi: 10.1007/978-3-540-36998-1_25.

[30] S. Barone, A. Paoli, and A. V. Razionale, ‘Assessment of chronic wounds by three-dimensional optical imaging based on integrating geometrical, chromatic, and thermal data’, *Proc. Inst. Mech. Eng. [H]*, vol. 225, no. 2, pp. 181–193, Feb. 2011, doi: 10.1243/09544119JEIM705.

[31] S. M. Ayaz, D. Khan, and M. Y. Kim, ‘Three-Dimensional Registration for Handheld Profiling Systems Based on Multiple Shot Structured Light’, *Sensors*, vol. 18, no. 4, Art. no. 4, Apr. 2018, doi: 10.3390/s18041146.

[32] J. M. Savage and S. Jeffery, ‘Use of 3D photography in complex-wound assessment’, *J. Wound Care*, vol. 22, pp. 156–60, Mar. 2013, doi: 10.12968/jowc.2013.22.3.156.

[33] C. Villa, M. J. Flies, and C. Jacobsen, ‘Forensic 3D documentation of bodies: Simple and fast procedure for combining CT scanning with external photogrammetry data’, *J. Forensic Radiol. Imaging*, vol. 10, pp. 47–51, Sep. 2017, doi: 10.1016/j.jofri.2017.08.003.

[34] C.-W. Chang, M.-C. Hsieh, I.-W. Lin, R.-F. Chen, Y.-R. Kuo, and S.-S. Lee, ‘Accreditation of the handheld 3-dimensional scanner and conventional photo images for area measurement’, *Medicine (Baltimore)*, vol. 103, no. 6, p. e35376, Feb. 2024, doi: 10.1097/MD.0000000000035376.

[35] S. Boersma, F. Heuvel, A. Cohen, and R. Scholtens, ‘Photogrammetric wound measurement with a three-camera vision system’, *IAPRS*, vol. XXXIII, Jan. 2000.

[36] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, ‘NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis’, Aug. 03, 2020, *arXiv*: arXiv:2003.08934. doi: 10.48550/arXiv.2003.08934.

[37] C. Rother, V. Kolmogorov, and A. Blake, ‘“GrabCut”: interactive foreground extraction using iterated graph cuts’, *ACM Trans. Graph.*, vol. 23, no. 3, pp. 309–314, Aug. 2004, doi: 10.1145/1015706.1015720.

[38] G. Scebba *et al.*, ‘Detect-and-segment: A deep learning approach to automate wound image segmentation’, *Inform. Med. Unlocked*, vol. 29, p. 100884, Jan. 2022, doi: 10.1016/j.imu.2022.100884.

[39] M. Adnan *et al.*, ‘An Automatic Wound Detection System Empowered by Deep Learning’, *J. Phys. Conf. Ser.*, vol. 2547, no. 1, p. 012005, Jul. 2023, doi: 10.1088/1742-6596/2547/1/012005.

[40] H. Carrión, M. Jafari, M. D. Bagood, H. Yang, R. R. Isseroff, and M. Gomez, ‘Automatic wound detection and size estimation using deep learning algorithms’, *PLoS Comput. Biol.*, vol. 18, no. 3, p. e1009852, Mar. 2022, doi: 10.1371/journal.pcbi.1009852.

[41] D. M. Anisuzzaman, Y. Patel, J. A. Niezgoda, S. Gopalakrishnan, and Z. Yu, ‘A Mobile App for Wound Localization Using Deep Learning’, *IEEE Access*, vol. 10, pp. 61398–61409, 2022, doi: 10.1109/ACCESS.2022.3179137.

[42] N. Curti *et al.*, ‘Effectiveness of Semi-Supervised Active Learning in Automated Wound Image Segmentation’, *Int. J. Mol. Sci.*, vol. 24, no. 1, Art. no. 1, Jan. 2023, doi: 10.3390/ijms24010706.

[43] D. M. Anisuzzaman, Y. Patel, B. Rostami, J. Niezgoda, S. Gopalakrishnan, and Z. Yu, ‘Multi-modal Wound Classification using Wound Image and Location by Deep Neural Network’, *Sci. Rep.*, vol. 12, no. 1, p. 20057, Nov. 2022, doi: 10.1038/s41598-022-21813-0.

[44] T. Cheng, L. Song, Y. Ge, W. Liu, X. Wang, and Y. Shan, ‘YOLO-World: Real-Time Open-Vocabulary Object Detection’, Feb. 22, 2024, *arXiv*: arXiv:2401.17270. Accessed: Jun. 06, 2024. [Online]. Available: http://arxiv.org/abs/2401.17270

[45] C. Boissin *et al.*, ‘Development and evaluation of deep learning algorithms for assessment of acute burns and the need for surgery’, *Sci. Rep.*, vol. 13, no. 1, p. 1794, Jan. 2023, doi: 10.1038/s41598-023-28164-4.

[46] M. Goyal, N. D. Reeves, S. Rajbhandari, N. Ahmad, C. Wang, and M. H. Yap, ‘Recognition of ischaemia and infection in diabetic foot ulcers: Dataset and techniques’, *Comput. Biol. Med.*, vol. 117, p. 103616, Feb. 2020, doi: 10.1016/j.compbiomed.2020.103616.

[47] ‘Download - DatabaseOfWoundsWebsite’. Accessed: Jun. 06, 2024. [Online]. Available: https://chronicwounddatabase.eu/Download

[48] ‘Pictures of wounds and surgical wound dressings’. Accessed: Jun. 06, 2024. [Online]. Available: https://www.medetec.co.uk/files/medetec-image-databases.html

[49] D. Buschi *et al.*, ‘Automated Wound Image Segmentation: Transfer Learning from Human to Pet via Active Semi-Supervised Learning’, *Animals*, vol. 13, no. 6, Art. no. 6, Jan. 2023, doi: 10.3390/ani13060956.

[50] R. Wu, D. Li, and C. Zhang, ‘Semi-supervised Medical Image Segmentation via Query Distribution Consistency’, arXiv.org. Accessed: Jun. 06, 2024. [Online]. Available: https://arxiv.org/abs/2311.12364v1

[51] ‘Semi-Supervised Learning: Techniques & Examples [2023]’. Accessed: Jun. 06, 2024. [Online]. Available: https://www.v7labs.com/blog/semi-supervised-learning-guide, https://www.v7labs.com/blog/semi-supervised-learning-guide

[52] M. Zhai, X. Xiang, N. Lv, and X. Kong, ‘Optical flow and scene flow estimation: A survey’, *Pattern Recognit.*, vol. 114, p. 107861, Jun. 2021, doi: 10.1016/j.patcog.2021.107861.

[53] J. Li, J. Zhao, S. Song, and T. Feng, ‘Unsupervised Joint Learning of Depth, Optical Flow, Ego-motion from Video’, May 30, 2021, *arXiv*: arXiv:2105.14520. doi: 10.48550/arXiv.2105.14520.

[54] J. M. Kang, Z. Sjanic, and G. Hendeby, ‘Optical Flow Revisited: how good is dense deep learning based optical flow?’, in *2023 IEEE Symposium Sensor Data Fusion and International Conference on Multisensor Fusion and Integration (SDF-MFI)*, Nov. 2023, pp. 1–6. doi: 10.1109/SDF-MFI59545.2023.10360502.

[55] Y. Matsushita, T. Yamaguchi, and H. Harada, ‘Object tracking using virtual particles driven by optical flow and Kalman filter’, in *2019 19th International Conference on Control, Automation and Systems (ICCAS)*, Oct. 2019, pp. 1064–1069. doi: 10.23919/ICCAS47443.2019.8971703.

[56] Y. Chen, D. Zhao, and H. Li, ‘Deep Kalman Filter with Optical Flow for Multiple Object Tracking’, in *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*, Oct. 2019, pp. 3036–3041. doi: 10.1109/SMC.2019.8914078.

[57] W. Bao, X. Zhang, L. Chen, and Z. Gao, ‘KalmanFlow 2.0: Efficient Video Optical Flow Estimation via Context-Aware Kalman Filtering’, *IEEE Trans. Image Process.*, vol. 28, no. 9, pp. 4233–4246, Sep. 2019, doi: 10.1109/TIP.2019.2903656.

[58] J. Hur and S. Roth, ‘Optical Flow Estimation in the Deep Learning Age’, in *Modelling Human Motion: From Human Perception to Robot Design*, N. Noceti, A. Sciutti, and F. Rea, Eds., Cham: Springer International Publishing, 2020, pp. 119–140. doi: 10.1007/978-3-030-46732-6_7.

[59] B. Zheng, X. Xu, Y. Dai, and Y. Lu, ‘Object Tracking Algorithm Based on Combination of Dynamic Template Matching and Kalman Filter’, in *2012 4th International Conference on Intelligent Human-Machine Systems and Cybernetics*, Aug. 2012, pp. 136–139. doi: 10.1109/IHMSC.2012.129.

[60] E. Karami, S. Prasad, and M. Shehata, ‘Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images’, Oct. 07, 2017, *arXiv*: arXiv:1710.02726. doi: 10.48550/arXiv.1710.02726.

[61] H. Bay, T. Tuytelaars, and L. Van Gool, ‘SURF: Speeded Up Robust Features’, in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Heidelberg: Springer, 2006, pp. 404–417. doi: 10.1007/11744023_32.

[62] D. G. Lowe, ‘Distinctive Image Features from Scale-Invariant Keypoints’, *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, Nov. 2004, doi: 10.1023/B:VISI.0000029664.99615.94.

[63] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, ‘ORB: An efficient alternative to SIFT or SURF’, in *2011 International Conference on Computer Vision*, Barcelona, Spain: IEEE, Nov. 2011, pp. 2564–2571. doi: 10.1109/ICCV.2011.6126544.

[64] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, ‘BRIEF: Binary Robust Independent Elementary Features’, presented at the Eur. Conf. Comput. Vis., Sep. 2010, pp. 778–792. doi: 10.1007/978-3-642-15561-1_56.

[65] Y.-C. Su *et al.*, ‘2.5D visual relationship detection’, *Comput. Vis. Image Underst.*, vol. 224, p. 103557, Nov. 2022, doi: 10.1016/j.cviu.2022.103557.

[66] N. Sharp and M. Ovsjanikov, ‘PointTriNet: Learned Triangulation of 3D Point Sets’, in *Computer Vision – ECCV 2020*, A. Vedaldi, H. Bischof, T. Brox, and J.-M. Frahm, Eds., Cham: Springer International Publishing, 2020, pp. 762–778. doi: 10.1007/978-3-030-58592-1_45.

[67] Z. Zhang, ‘Epipolar Geometry’, in *Computer Vision: A Reference Guide*, K. Ikeuchi, Ed., Cham: Springer International Publishing, 2021, pp. 387–400. doi: 10.1007/978-3-030-63416-2_128.

[68] Y.-J. Zhang, ‘Camera Calibration’, in *3-D Computer Vision: Principles, Algorithms and Applications*, Y.-J. Zhang, Ed., Singapore: Springer Nature, 2023, pp. 37–65. doi: 10.1007/978-981-19-7580-6_2.

[69] S. Jiang, C. Jiang, and W. Jiang, ‘Efficient structure from motion for large-scale UAV images: A review and a comparison of SfM tools’, *ISPRS J. Photogramm. Remote Sens.*, vol. 167, pp. 230–251, Sep. 2020, doi: 10.1016/j.isprsjprs.2020.04.016.

[70] Y. Hong, J. Zhang, B. Jiang, Y. Guo, L. Liu, and H. Bao, ‘StereoPIFu: Depth Aware Clothed Human Digitization via Stereo Vision’, presented at the Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 535–545. Accessed: Jun. 17, 2024. [Online]. Available: https://openaccess.thecvf.com/content/CVPR2021/html/Hong_StereoPIFu_Depth_Aware_Clothed_Human_Digitization_via_Stereo_Vision_CVPR_2021_paper.html

[71] D. Gorjup, J. Slavič, A. Babnik, and M. Boltežar, ‘Still-camera multiview Spectral Optical Flow Imaging for 3D operating-deflection-shape identification’, *Mech. Syst. Signal Process.*, vol. 152, p. 107456, May 2021, doi: 10.1016/j.ymssp.2020.107456.

[72] Q. Wang *et al.*, ‘Tracking Everything Everywhere All at Once’, Sep. 12, 2023, *arXiv*: arXiv:2306.05422. doi: 10.48550/arXiv.2306.05422.

[73] N. Tumanyan, A. Singer, S. Bagon, and T. Dekel, ‘DINO-Tracker: Taming DINO for Self-Supervised Point Tracking in a Single Video’, Mar. 21, 2024, *arXiv*: arXiv:2403.14548. doi: 10.48550/arXiv.2403.14548.

[74] ‘OpenCV: OpenCV modules’. Accessed: Jun. 24, 2024. [Online]. Available: https://docs.opencv.org/4.x/

[75] O. Ronneberger, P. Fischer, and T. Brox, ‘U-Net: Convolutional Networks for Biomedical Image Segmentation’, May 18, 2015, *arXiv*: arXiv:1505.04597. Accessed: Apr. 22, 2024. [Online]. Available: http://arxiv.org/abs/1505.04597

[76] ‘Get actor translation, rotation, scale - Simulink’. Accessed: Jun. 15, 2024. [Online]. Available: https://www.mathworks.com/help/vdynblks/ref/simulation3dactortransformget.html

[77] E. Malis and M. Vargas, ‘Deeper understanding of the homography decomposition for vision-based control’, Jan. 2007.

[78] ‘Hypodermis (Subcutaneous Tissue): Function & Structure’, Cleveland Clinic. Accessed: Jun. 15, 2024. [Online]. Available: https://my.clevelandclinic.org/health/body/21902-hypodermis-subcutaneous-tissue

[79] ‘What is NumPy? — NumPy v2.0 Manual’. Accessed: Jun. 26, 2024. [Online]. Available: https://numpy.org/doc/stable/user/whatisnumpy.html

[80] ‘Six-axis Robotic Arm Industrial robot collaborative robot’, seedrobotarm. Accessed: Jun. 16, 2024. [Online]. Available: https://seedrobotarm.com/product/six-axis-robotic-arm-industrial-robot-collaborative-robot/

Appendix A

Code for the main processing pipeline.

```
import time
import math
import numpy as np
import struct

import cv2

import serial
import threading
import queue

import traceback

np.object = object
np.bool = bool
np.int = int

from tensorflow import keras
from tensorflow.keras import layers


def build_unet_model(input_shape):
    """Initialize model.

    U-Net model architecture.

    Args:
        input_shape (tuple): input image size: (height, width, channels)

    Returns:
        model
    """
    inputs = keras.Input(shape=input_shape)

    # Encoder
    conv1 = layers.Conv2D(
        64, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(inputs)
    conv1 = layers.Conv2D(
        64, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv1)
```

```
    pool1 = layers.MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(pool1)
    conv2 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv2)
    pool2 = layers.MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = layers.Conv2D(
        256, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(pool2)
    conv3 = layers.Conv2D(
        256, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv3)
    pool3 = layers.MaxPooling2D(pool_size=(2, 2))(conv3)

    # Bottleneck
    conv4 = layers.Conv2D(
        512, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(pool3)
    conv4 = layers.Conv2D(
        512, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv4)

    # Decoder
    up1 = layers.UpSampling2D((2, 2))(conv4)
    concat1 = layers.Concatenate(axis=-1)([conv3, up1])
    conv5 = layers.Conv2D(
        256, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(concat1)
    conv5 = layers.Conv2D(
        256, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv5)
    up2 = layers.UpSampling2D((2, 2))(conv5)

    concat2 = layers.Concatenate(axis=-1)([conv2, up2])
    conv6 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(concat2)
    conv6 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv6)
    up3 = layers.UpSampling2D((2, 2))(conv6)
```

```
    concat3 = layers.Concatenate(axis=-1)([conv1, up3])
    conv7 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(concat3)
    conv7 = layers.Conv2D(
        128, 3, activation='relu', kernel_initializer='he_normal', padding='same'
    )(conv7)

    outputs = layers.Conv2D(1, 1, activation='sigmoid')(
        conv7
    )  # binary segmentation output

    model = keras.Model(inputs=inputs, outputs=outputs)
    return model


### Loading the wound segmentation model
model = build_unet_model((128, 128, 1))  # trained model size
model.load_weights('path to model')


### Initial 6-DOF robot arm position
J6 = 0  # J6 angle (wrist)
J5 = 0  # J5 angle (elbow)
X = 203  # robot's X position
Y = 163  # robot's Y position
Z = 30  # robot's Z position


def format_message(value):
    """Convert float to a hexadecimal string.

    1.  Format float numbers into the IEEE 754 binary32 single-precision format.
    2.  Convert every byte to a lowercase hexadecimal string prefixed with “0x”.
    3.  Convert the hexadecimal representation to an integer with 16 bases

    Args:
        value (float): input float number

    Returns:
        list: list of hexadecimal
    """
    return format_message(value)
```

```
def send_message_G1(serial_port, j6, j5, x, y, z, speed):
    """Send G1 code instructions.

    Args:
        serial_port (serial): connector
        j6, j5, x, y, z (float): new position
        speed (float): selected speed (mm/min)
    """
    # Pack instruction frame
    instruction = [0] * 48  # frame consists of 48 bytes
    instruction[0] = [238]  # frame head - G1 code equivalent
    instruction[1] = '1'.encode()
    instruction[2] = 1
    # every code takes 4 bytes
    instruction[3:7] = format_message(x)  # x
    instruction[7:11] = format_message(y)  # y
    instruction[11:15] = format_message(z)  # z
    instruction[15:19] = format_message(j5)  # J5
    instruction[23:27] = format_message(j6)  # J6
    instruction[43:47] = format_message(speed)  # speed
    instruction[47] = 239  # frame tail

    # convert instructions to bytes
    complete_instruction = bytes(instruction[0]) + instruction[1] +
bytes(instruction[2:])
    serial_port.write(complete_instruction)  # send instructions


def homing(serial_port, j6=0, j5=0, x=203, y=163, z=30, speed=1000):
    """Homing the robot.

    After the main process is done, homing to the initial position.

    Args:
        serial_port (serial): connector
        j6, j5, x, y, z (float): home position
        speed (float): selected speed (mm/min). Defaults to 1000.
    """
    print('HOMING')
    send_message_G1(serial_port, j6, j5, x, y, z, speed)
    time.sleep(3)  # wait for robot to home
    print('HOMED')


def find_wound(gray_frame, model, image_height=720, image_width=1280):
```

```
    """Find a wound in image and segment it.

    Args:
        gray_frame: frame in grayscale
        model: segmentation model
        image_height (int): height of the image. Defaults to 720.
        image_width (int): width of the image. Defaults to 1280.

    Returns:
        np.ndarray: segmentation mask.
    """
    image = cv2.resize(gray_frame, (128, 128))  # resize image to march model
input
    image = image / 255  # normalize the image [0 ... 1]
    mask = np.array(model(np.array([image])))[0]  # predict segmentation mask

    resized_mask = cv2.resize(
        mask, (image_height, image_width)
    )  # return original dimensions
    threshold = 0.7  # initial threshold

    ### Decrease the threshold until we find the mask or the threshold reaches
0.1 value
    while threshold > 0.1:
        # pixel equals to 0 if it is lower than the threshold, 1 otherwise
        filtered_mask = np.where(resized_mask > threshold)

        if len(filtered_mask[0]) == 0:  # if no pixel has value
            threshold -= 0.1  # decrease the threshold
        else:
            break  # otherwise return filtered_mask
    return filtered_mask


def crop_ROI(
    gray_frame,
    mask,
    previous_ROI=None,
    previous_frame_gray=None,
    border=5,
    image_height=720,
    image_width=1280,
    initial_run_flag=False,
):
    """Crop ROI based on segmented mask.
```

```
    Args:
        gray_frame: current frame in grayscale
        mask: segmented wound mask
        previous_ROI: previous ROI coordinates
        previous_frame_gray: previous frame in grayscale
        border (int, optional): border around ROI (px). Defaults to 5.
        image_height (int): height of the image. Defaults to 720.
        image_width (int): width of the image. Defaults to 1280.
        initial_run_flag: points if now is the first program run. Defaults to
False.

    Returns:
        cropped current and previous frames in grayscale, new ROI coordinates
    """
    # find bounding box coordinates of the mask
    xmin, xmax, ymin, ymax = mask[0].min(), mask[0].max(), mask[1].min(),
mask[1].max()

    ### If it's an initial program run, set current values as previous ones
    if not previous_ROI:
        initial_run_flag = True
        previous_ROI = xmin, xmax, ymin, ymax

    xmin_previous, xmax_previous, ymin_previous, ymax_previous = previous_ROI
    previous_ROI = (
        xmin,
        xmax,
        ymin,
        ymax,
    )  # update previous coordinates with the new ROI

    # select min max values between current and previous masks
    min_x, max_x, min_y, max_y = (
        min(xmin_previous, xmin),
        max(xmax_previous, xmax),
        min(ymin_previous, ymin),
        max(ymax_previous, ymax),
    )

    # add border, but keep it inside of the image
    if border:
        min_x, max_x, min_y, max_y = (
            max(0, min_x - border),
            min(image_width, max_x + border),
```

```
            max(0, min_y - border),
            min(image_height, max_y + border),
        )

    # crop ROI on a current frame
    cropped_gr = gray_frame.copy()[
        min_x:max_x,
        min_y:max_y,
    ]

    # crop ROI on a previous frame
    if initial_run_flag:
        previous_frame_gray = cropped_gr.copy()
    else:
        previous_frame_gray = previous_frame_gray.copy()[
            min_x:max_x,
            min_y:max_y,
        ]
    ### Apply Bilateral filter on both ROIs
    return (
        cv2.bilateralFilter(cropped_gr, 10, 75, 750),
        cv2.bilateralFilter(previous_frame_gray, 10, 75, 750),
        previous_ROI,
    )


def track(
    current_ROI,
    previous_ROI,
    feature_extractor_SIFT,
    key_points_current=None,
    descriptor_current=None,
    destination_points=None,
    feature_matcher_FLANN=None,
):
    """Track the wound and estimate its displacement.

    Args:
        current_ROI: current ROI
        previous_ROI: previous ROI to compare with
        feature_extractor_SIFT: feature extractor algorithm (SIFT in our case)
        feature_matcher_FLANN: feature matcher algorithm (FLANN in our case)
        key_points_current: previously calculated key points in the previous
frame
```

```
        descriptor_current: previously calculated descriptors in the previous
frame
        destination_points: previously calculated destination points in the
previous frame

    Returns:
        source points, X-axis displacement, Y-axis displacement, key points,
descriptors,
        and destination points of the current frame
    """
    ### if it is not the first run, use previously calculated values as values
for the previous_ROI,
    ### otherwise calculate them
    if key_points_current:
        key_points_previous, descriptor_previous = (
            key_points_current,
            descriptor_current,
        )
        descriptor_previous = np.float32(descriptor_previous)
        source_points = destination_points
    else:
        key_points_previous, descriptor_previous = (
            feature_extractor_SIFT.detectAndCompute(previous_ROI, None)
        )
        descriptor_previous = np.float32(descriptor_previous)

    # extract features from the current ROI
    key_points_current, descriptor_current =
feature_extractor_SIFT.detectAndCompute(
        current_ROI, None
    )
    descriptor_current = np.float32(descriptor_current)

    ### Match features, apply Lowe's ratio to filter bad matches
    matches = feature_matcher_FLANN.knnMatch(
        descriptor_previous, descriptor_current, k=2
    )
    good_matches = []
    for m, n in matches:
        if m.distance < 0.8 * n.distance:
            good_matches.append(m)

    ### Find features old and new coordinates
    source_points = np.float32(
        [key_points_previous[m.queryIdx].pt for m in good_matches]
```

```
    ).reshape(-1, 1, 2)
    destination_points = np.float32(
        [key_points_current[m.trainIdx].pt for m in good_matches]
    ).reshape(-1, 1, 2)

    ### Calculate the median displacement
    moved_X = np.median(destination_points[:, 0, 0] - source_points[:, 0, 0])
    moved_Y = np.median(destination_points[:, 0, 1] - source_points[:, 0, 1])

    return (
        source_points,
        moved_X,
        moved_Y,
        key_points_current,
        descriptor_current,
        destination_points,
    )


def rotation(source_points, destination_points):
    """Estimate rotation.

    Args:
        source_points: source points
        destination_points: destination points

    Returns:
        rotation in degrees
    """
    M, _ = cv2.findHomography(
        source_points, destination_points, cv2.RANSAC, ransacReprojThreshold=3.0
    )
    return np.degrees(np.arctan2(M[1, 0], M[0, 0]))


def convert_to_world_values(x, y, scale_factor):
    """Apply scale factor to the displacements.

    Args:
        x: X value in pixels
        y: Y value in pixels
        scale_factor: scaling

    Returns:
        World coordinates for X and Y in mm
```

```
    """
    return x * scale_factor, y * scale_factor


def move_arm(new_J6, new_J5, new_x, new_y, new_z, serial_port, J6, J5, X, Y, Z):
    """Move robot to designated coordinates.

    Args:
        new_J6: degrees to add
        new_J5: degrees to add
        new_x: distance to add
        new_y: distance to add
        new_z: distance to add
        serial_port: connector
        J6: current J6 angle
        J5: current J5 angle
        X: current X position
        Y: current Y position
        Z: current Z position

    Returns:
        New angles and positions
    """
    J6 += new_J6
    J5 += new_J5
    X += new_x
    Y += new_y
    Z += new_z

    send_message_G1(serial_port, J6, J5, X, Y, Z, speed=2500)

    return J6, J5, X, Y, Z


def capture_frames(
    camera_ID,
    moved_queue,
    event,
    feature_extractor_SIFT,
    feature_matcher_FLANN,
    scale_factor,
):
    """Capture and process frame from the camera in a separate thread.

    Args:
```

```
        camera_ID: current camera (top/side/bottom)
        moved_queue: placeholder for displacement values
        event: flagger to other threads
        feature_extractor_SIFT: feature extractor algorithm
        feature_matcher_FLANN: feature matcher algorithm
        scale_factor: camera calibration
    """
    ### Initialise variables
    previous_ROI = None
    keypoints, descriptor, destination_points = None, None, None
    ### Read every 30th frame
    frame_to_skip = 30
    current_frame_index = -1

    ### Open camera
    camera = cv2.VideoCapture(camera_ID, cv2.CAP_DSHOW)
    camera.set(cv2.CAP_PROP_BUFFERSIZE, 0)
    camera.set(cv2.CAP_PROP_FRAME_WIDTH, 1280)
    camera.set(cv2.CAP_PROP_FRAME_HEIGHT, 720)

    ### Read first frame and convert it to grayscale
    _, previous_ROI = camera.read()
    previous_frame_gray = cv2.cvtColor(previous_ROI, cv2.COLOR_BGR2GRAY)

    image_width, image_height = previous_frame_gray.shape[:2]  # get width and
height

    while True:
        retrive, frame = camera.read()  # read frame

        ### If camera is closed, stop reading frames and send commands to the
robot to home
        if not retrive:
            print("STOPING...")
            moved_queue.put(
                (None, None, None)
            )  # send None types instead of real values
            event.set()  # set flag that the current thread finished iteration
            camera.release()  # stop camera processing
            print('STOPPED')
            break

        ### Skip frames, do not process skipped
        current_frame_index += 1
        if current_frame_index % frame_to_skip != 0:
```

```
            continue

        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  # convert to gray

        ### Process frame
        try:
            mask = find_wound(
                frame_gray,
                model,
                cid=camera_ID,
                image_height=image_height,
                image_width=image_width,
            )

            current_ROI, previous_frame_gray, previous_ROI = crop_ROI(
                frame_gray,
                mask,
                previous_ROI,
                previous_frame_gray,
                image_height=image_height,
                image_width=image_width,
            )

            source_points, x, y, keypoints, descriptor, destination_points =
track(
                current_ROI,
                previous_frame_gray,
                feature_extractor_SIFT,
                keypoints,
                descriptor,
                destination_points,
                feature_matcher_FLANN,
                mask,
            )
            joint_rotation = rotation(source_points, destination_points)
            x, y = convert_to_world_values(x, y, scale_factor)

            # put the frame in the queue with the movements values
            moved_queue.put((joint_rotation, x, y))

            # set the event to signal that a frame has been processed
            event.set()

            # reassign current frame as a previous one
            previous_frame_gray = frame_gray
```

```
        except Exception as e:
            # if there any runtime error in the code, log the error
            print(f"ERROR : {e}  {traceback.format_exc()}")


def process_arm(
    top_camera_queue,
    bottom_camera_queue,
    top_camera_event,
    bottom_camera_event,
    serial_port,
    J6, J5, X, Y, Z,
):
    """Robot arm manipulations.

    Args:
        top_camera_queue: results from top camera thread
        bottom_camera_queue: results from bottom camera thread
        top_camera_event: top camera's flag
        bottom_camera_event: bottom camera's flag
        serial_port: connector
        J6: current J6 position
        J5:  current J5 position
        X:  current X position
        Y:  current Y position
        Z:  current Z position
    """
    ### Endless loop, until the condition applies
    while True:
        ### Wait for both events to be set
        top_camera_event.wait()
        bottom_camera_event.wait()

        ### Get frames from both queues
        new_J6, new_x, new_y = top_camera_queue.get()
        new_J5, _, new_z = bottom_camera_queue.get()

        ### If queues return None's, stop all the processes
        if (new_J6 == None and new_x == None and new_y == None) and (
            new_J5 == None and new_z == None
        ):
            time.sleep(10)  # wait 'till robot finishes the last movement
            homing(serial_port)
            serial_port.close()  # close connector
            print("PRINTER STOPPED")
```

```
            break

        J6, J5, X, Y, Z = move_arm(
            new_J6, new_J5, new_x, new_y, new_z, serial_port, J6, J5, X, Y, Z
        )

        ### Clear the events for the next iteration
        top_camera_event.clear()
        bottom_camera_event.clear()


### Script starts from here
if __name__ == '__main__':
    # initialise connector from the host computer to the robot
    serial_port = serial.Serial(
        'COM3',
        baudrate=115200,
        timeout=None,
        bytesize=8,
        stopbits=serial.STOPBITS_ONE,
        parity=serial.PARITY_NONE,
    )

    # choose feature extraction and matching algorithms
    feature_extractor_SIFT = cv2.SIFT_create(nfeatures=100)
    index_params = dict(algorithm=1, trees=4)
    search_params = dict(checks=10)
    feature_matcher_FLANN = cv2.FlannBasedMatcher(index_params, search_params)
    # OR try BF matcher instead of FLANN
    # feature_matcher_BF = cv2.BFMatcher(cv2.NORM_L2, crossCheck=False)

    # calibration
    scale_factor = 0.45  # mm/px

    ### Initialise queue per camera. Each queue has 3 elements
    top_camera = queue.Queue(3)
    bottom_camera = queue.Queue(3)

    ### Initialise event flags per camera
    top_camera_event = threading.Event()
    bottom_camera_event = threading.Event()

    # initialise thread for the top camera
    top_camera_thread = threading.Thread(
        target=capture_frames,
```

```
        args=(
            0,  # top camera ID
            top_camera,
            top_camera_event,
            feature_extractor_SIFT,
            feature_matcher_FLANN,
            scale_factor,
        ),
    )
    # initialise thread for the bottom camera
    bottom_camera_thread = threading.Thread(
        target=capture_frames,
        args=(
            2,  # bottom camera ID
            bottom_camera,
            bottom_camera_event,
            feature_extractor_SIFT,
            feature_matcher_FLANN,
            scale_factor,
        ),
    )
    # initialise thread for the robot
    robot_thread = threading.Thread(
        target=process_arm,
        args=(
            top_camera,
            bottom_camera,
            top_camera_event,
            bottom_camera_event,
            serial_port,
            J6, J5, X, Y, Z,
        ),
    )

    ### Start threads
    top_camera_thread.start()
    bottom_camera_thread.start()
    robot_thread.start()

    ### Finish threads
    top_camera_thread.join()
    bottom_camera_thread.join()
    robot_thread.join()
```