MCAST

Tracking and Pose Estimation of a Robotic System using Computer Vision

Karl Spiteri

Supervisor: Dr Ing. Jeremy Scerri

September 2023

A dissertation submitted to the Institute of Engineering and Transport in partial fulfilment of the requirements for the degree Bachelor of Engineering (Honours) in Control and Electronics Engineering.

Authorship Statement

This dissertation is based on the results of research carried out by myself, is my own composition, and has not been previously presented for any other certified or uncertified qualification.

The research was carried out under the supervision of Dr Ing. Jeremy Scerri.

15th September 2023

_____

Copyright Statement

In submitting this dissertation to the MCAST Institute of Engineering and Transport, I understand that I am giving permission for it to be made available for use in accordance with the regulations of MCAST and the College Library.

15th September 2023

_____

Acknowledgements

Thank You to my supervisor, friends, and family who supported me throughout my thesis. Their support, encouragement, and faith in me were significant. Their support helped me overcome obstacles and hurdles I encountered.

Abstract

This thesis presents a comprehensive approach to tracking and pose estimation of a robotic arm using computer vision techniques. The primary goal of the research is to develop a real-time computer vision system capable of tracking the robotic arm, estimating its joint angles, and providing a live virtual prototype of the robotic arm.

The development process involved several key steps. Firstly, a tracking system was constructed to enable robust custom object tracking. This involved implementing algorithms for feature detection, matching, and tracking to accurately follow the movement of the targeted object. Additionally, a marker generator was built to create unique markers tailored to the robotic arm. These markers were designed to provide distinct visual features for reliable pose estimation.

To ensure precise marker placement on the robotic arm, 3D printed custom designed placers were designed using CAD techniques. These placers were custom designed to fit the specific dimensions and geometry of the robotic arm, facilitating accurate marker positioning. The markers were strategically placed on the arm to enable precise joint angle estimation.

Marker data acquisition was a crucial step in the process. The computer vision system read the marker data and extracted the necessary matrices for subsequent calculations. These matrices contained essential information about the position, orientation, and motion of the markers, enabling further analysis.

Using the data from the markers further calculations would lead to achieving the joint angles of the robotic arm. Then kinematic techniques can be applied to construct a live virtual prototype of the robotic arm in real-time therefore having a visual representation.

Contents

List of Figures

Glossary of Terms

API: Application Programming Interface - A set of rules and protocols that allows different software applications to communicate and interact with each other.

CAD: Computer-Aided Design - The use of computer software to create and modify designs, typically for engineering, architectural, or manufacturing purposes.

CNN: Convolutional Neural Networks - A class of deep neural networks designed to process structured grid-like data, such as images or video, using convolutional layers for feature extraction.

CSV: Comma-Separated Values - A simple file format used to store tabular data, where each value is separated by a comma. It is commonly used for data interchange between different software applications.

DH: Denavit-Hartenberg - A convention used in robotics to represent the geometric transformations and kinematics of robotic manipulator systems.

MAP: Mean Average Precision - A metric commonly used to evaluate the accuracy of object detection and segmentation models by measuring the precision and recall of the predicted results.

RCNN: Region-based Convolutional Neural Networks - A family of object detection algorithms that generate region proposals and use convolutional neural networks to classify and refine the proposed regions.

RESNET: Residual Network - A deep neural network architecture known for its residual connections, which allow for training very deep networks by addressing the vanishing gradient problem.

SSDS: Single Shot Detectors - A type of object detection algorithm that predicts bounding boxes and class labels in a single pass through the network, making it efficient for real-time applications.

TFRECORD: TensorFlow Record - A file format used for storing large amounts of data in

YOLO: You Only Look Once - An object detection algorithm that processes the entire image in a single pass, making it efficient for real-time applications and known for its fast inference speed.

List of Abbreviations

API	Application Programming Interface
CAD	Computer-Aided Design
CNN	Convolutional Neural Networks
CSV	Comma-Separated Values
DH	Denavit-Hartenberg
MAP	Mean Average Precision
RCNN	Region-based Convolutional Neural Networks
RESNET	Residual Network
SSDS	Single Shot Detectors
TFRECORD	TensorFlow Record
YOLO	You Only Look Once

1. Introduction

In recent years, the remarkable advancements in robotics have empowered robotic systems to accomplish intricate tasks with unprecedented precision. Among the pivotal facets of this progress is the capacity for real-time tracking and position estimation of robotic arms, an essential element in optimizing their interaction with the environment.

This thesis endeavours to forge a path in the realm of robotics by creating a computer vision system saturated with the capability for real-time object tracking and the estimation of joint angles for a robotic arm. In addition to this, the system seamlessly assimilates the acquired data to generate a live virtual prototype of the robotic arm within the MATLAB environment, offering a visually intuitive representation of its structure and dynamic movements. Through the harmonious integration of principles from computer vision and robotics, this research aspires to amplify the capabilities of robotic arms and elevate their proficiency in engaging with the surrounding world.

This pursuit of innovation is not without its challenges. Achieving real-time object tracking mandates the deployment of robust algorithms that can adeptly navigate through occlusions, accommodate fluctuations in lighting conditions, and adapt to diverse object appearances. Likewise, the precision of joint angle estimation for the robotic arm necessitates the fusion of visual data with intricate kinematic models. Furthermore, the generation of a live virtual prototype

mandates the harmonious amalgamation of real-time pose estimation and visualization techniques.

In the face of these formidable challenges, this thesis adopts a systematic approach. A sophisticated tracking system is meticulously crafted, harnessing cutting-edge computer vision algorithms to facilitate feature detection, matching, and tracking. Augmentations and modifications, purposefully tailored for the robotic arm, are diligently engineered using CAD software. These modifications include custom-designed marker placers and mechanical enhancements, geared towards the attainment of precise and reliable pose estimation.

Central to the functioning of this system is the acquisition of marker data, an elemental process that informs the entire operation. The computer vision system adeptly interprets marker data, extracting essential matrices that encode vital information concerning the markers' spatial orientation, position, and motion. These matrices, in turn, serve as pivotal inputs for subsequent calculations, thus enabling the precise estimation of joint angles and the real-time projection of the robotic arm's pose.

The ramifications of this research are both profound and far-reaching. The development of a robust computer vision system for tracking and pose estimation of robotic arms holds the potential to revolutionize an array of domains, including industrial automation, healthcare, and exploration. By bestowing robotic arms with the ability to acutely perceive and interpret their surroundings, this research

heralds a new era of enhanced autonomy and efficiency, fostering possibilities for unparalleled collaboration and interaction between humans and robots.

In summation, this thesis presents an innovative paradigm in the arena of tracking and pose estimation for robotic arms through the judicious integration of computer vision techniques. The resultant system not only tracks objects and estimates joint angles but also provides a real-time, dynamic visualization of the arm's movement. In this convergence of computer vision and robotics, lies the promise of refining and expanding the utility of robotic arms.

2. Literature Review

2.1 Tracking Techniques in Computer Vision

The domain of computer vision, an amalgamation of computational and imaging processes, has witnessed an unprecedented surge in growth over the past few decades. Within this expansive domain, a central focus has been the tracking of objects or features within sequences of images or videos, a pursuit that has undergone remarkable transformation both in terms of methodologies employed and the achieved outcomes.

In the nascent stages of computer vision, object tracking predominantly relied on template matching. This method entailed sliding a template of the target object across each frame and seeking the optimal match. While conceptually straightforward, this approach incurred substantial computational costs and faltered when faced with objects undergoing substantial alterations in appearance, orientation, or scale. Moreover, early tracking methods grappled with the constraints of static backgrounds and limited object movement.

In the contemporary landscape, tracking techniques have evolved significantly, blending probabilistic frameworks, machine learning paradigms, and deep learning architectures. Algorithms such as Mean Shift, Kalman Filters, and Particle Filters elevated tracking to a realm of adaptability and robustness. The advent of deep learning further catalysed this evolution, with architectures like Single Shot Detectors (SSDs), Convolutional Neural Networks (CNNs), and You Only

Look Once (YOLO) models, leading the way in facilitating real-time, resource-efficient, and highly accurate object detection and tracking.

The profound strides made in tracking methodologies bear immense significance for robotic systems. Across diverse domains, from autonomous driving to warehouse automation, robots rely upon precise tracking to navigate, interact with objects, and even discern and interpret gestures. In environments where change is the only constant, the ability to robustly track entities empowers robots to make informed decisions, respond adeptly to unforeseen obstacles, and execute tasks with pinpoint accuracy.

In essence, the evolution of tracking techniques within the realm of computer vision signifies a transformative journey—from rudimentary pattern matching to sophisticated algorithms fuelled by artificial intelligence. As the demands on robotics continue to escalate in terms of precision and adaptability, the ongoing advancement of tracking technologies emerges as a linchpin in shaping the future of intelligent and responsive robotic systems.

2.1.1 Traditional Tracking Approaches

In the domain of position estimation, particularly within complex and unstructured environments, two prominent methodologies have surfaced as solutions of significance: the Kalman Filter and the Particle Filter. These are probabilistic algorithms designed to furnish precise tracking capabilities across various applications [1] [2].

The Kalman Filter has garnered acclaim for its optimal implementation of the Bayes estimator, particularly well-suited for linear systems characterized by Gaussian noise [1]. It operates through a recursive two-step estimation process, encompassing prediction and correction phases. The Kalman Filter necessitates the definition of a model and is notably proficient in scenarios where the system's behaviour adheres to linearity [1]. However, the application of the Kalman Filter becomes increasingly intricate when tasked with tracking a variable number of objects, compelling the incorporation of an association algorithm to ensure that the estimator receives the accurate input [1]. *Figure 1* highlights the improvement of the Kalman filter (Blue) when comparing the Internal Navigation System (INS) (Red) and the reference of the GPS (Green).

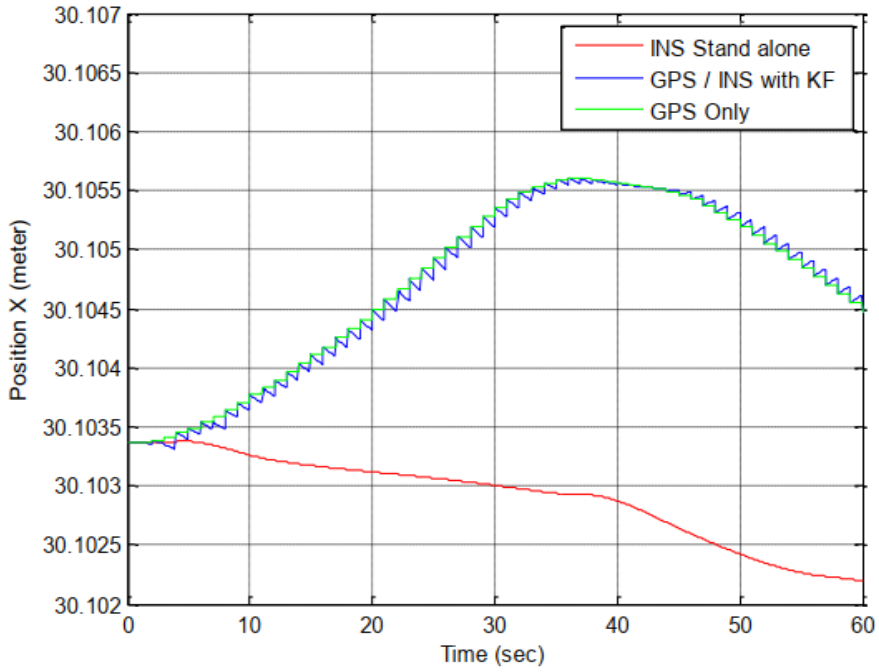

Figure 1: Kalman Filter (KF) Results [1].

Conversely, the Particle Filter presents a notably flexible approach, particularly well-suited for navigating the intricacies of non-linear and non-Gaussian

environments [2]. Unlike the Kalman Filter, the Particle Filter refrains from discretizing the state vector, endowing it with enhanced accuracy in its estimations [2]. This attribute also translates into a reduced computational load, rendering the Particle Filter eminently suitable for real-time estimation. A standout feature of the Particle Filter lies in its unique capability to encapsulate multiple estimation hypotheses within a single algorithm, a dimension that remains beyond the purview of the Kalman Filter [2]. As depicted in *Figure 2* Particle filter is used to for camera tracking where the weighted particle shows a potential location for the camera.

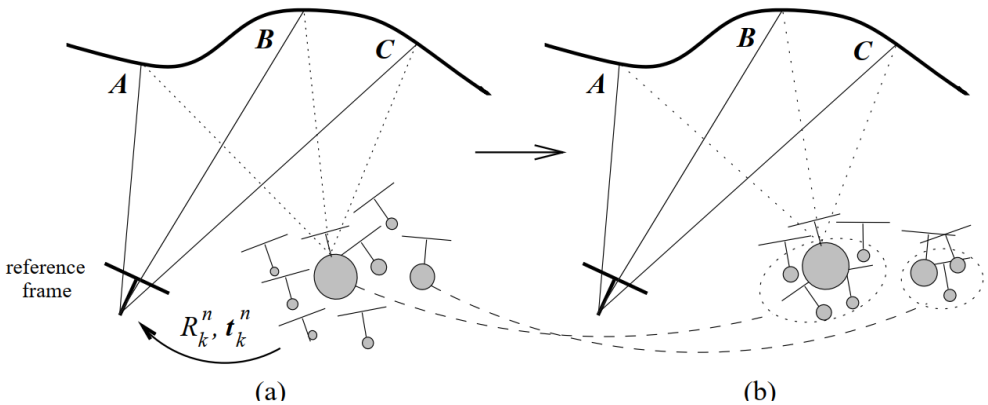

Figure 2: Particle Filter Approximation Results [2].

In a head-to-head comparison of the two filters within the context of a multi-object tracking application, the Particle Filter underwent extension and adaptation, supplemented by a clustering process to proficiently track a variable number of objects. In contrast, the Kalman Filter was deployed in conjunction with an association algorithm for each object under surveillance [3]. The outcomes gleaned from real-time execution within such applications unveiled distinct

advantages and limitations intrinsic to both algorithms. This could be observed in *Figure 3*.

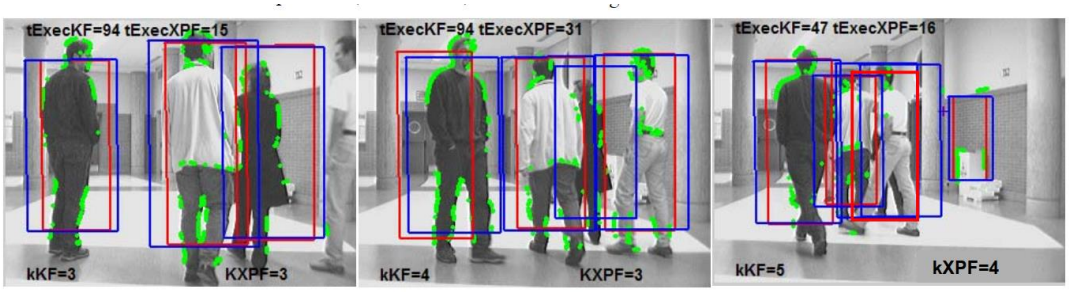

Figure 3: Comparison of Tracking Results Between Kalman Filter and Particle Filter [3].

2.1.2 Deep Learning-based Tracking Techniques

The advent of deep learning has sparked a revolution in the realm of object detection, with models such as the Single Shot MultiBox Detector (SSD), You Only Look Once (YOLO), and RetinaNet spearheading this transformative wave. These models have found extensive utility across diverse applications, including the realm of real-time object detection.

Findings from [4] illuminated a compelling connection among these models. RetinaNet, while achieving an impressive mean average precision (MAP) of 82.89%, exhibited a frames per second (FPS) performance that was merely one-third of that achieved by YOLO v3, posing challenges for real-time deployment. In contrast, SSD, though competent, did not excel in terms of both MAP and FPS metrics. YOLO v3, despite marginally lower MAP scores (80.69%), delivered a significant advantage in terms of detection speed. Furthermore, YOLO v3 demonstrated superior performance in detecting complex and challenging

samples, thus rendering it more amenable for deployment in contexts such as hospital equipment [4]. This could be observed in *Figure 4*.

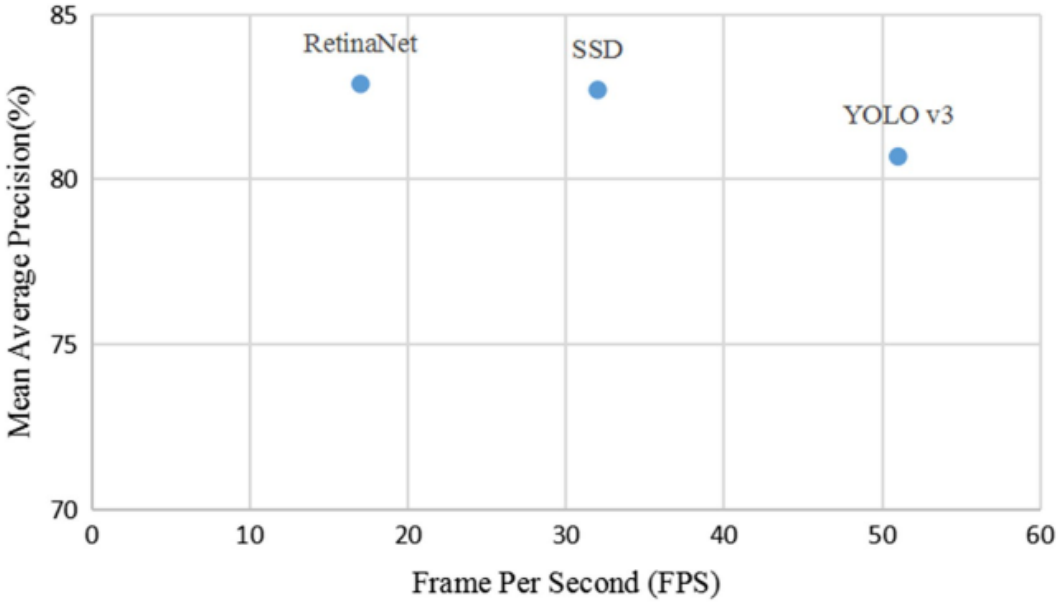

Figure 4: Performance of deep learning model [4].

An additional research endeavour delving into the comparison between SSD and YOLO centred its focus on the detection of outdoor urban advertising panels, subject to diverse environmental variabilities. This investigation underscored that SSD virtually eradicated False Positive cases, a desirable trait when analysing the content within the detected panels. Conversely, YOLO exhibited enhanced panel localization results, proficiently detecting a higher number of True Positive panels with augmented accuracy [5]. This could be seen in *Figure 5*.

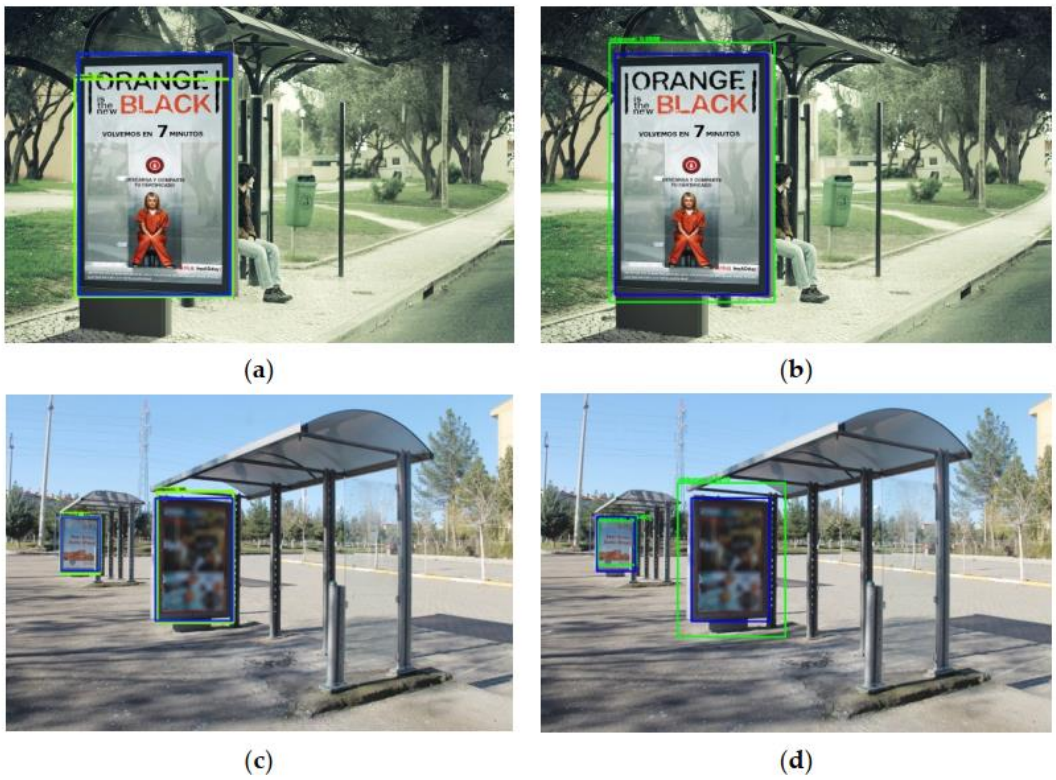

Figure 5: Qualitative detection between SSD and YOLOv3 [5].

2.2 Computer Vision Marker Based Pose Estimation in Robotics

In the domain of robotics, the precise determination of an object's pose, encompassing both its position and orientation within the surrounding space, stands as an indispensable prerequisite. The attainment of accurate pose estimation is the linchpin upon which a robot's ability to effectively interact with its environment hinges. Whether the objective entails object manipulation, obstacle avoidance, or intricate navigation within a complex milieu, precise pose estimation is paramount. Within this purview, computer vision-based techniques, particularly those grounded in marker-based methodologies, emerge as a prominent avenue for achieving this critical task.

Marker-based systems rely predominantly on the strategic placement of discernible markers upon a robot or the object under scrutiny. These markers typically manifest as 2D patterns or intricate 3D structures, meticulously crafted to facilitate effortless detection and recognition by cameras or other optical sensors. Upon successful detection, the spatial information gleaned from these markers undergoes a transformation, effectively translating into a comprehensive representation of the object's pose.

In the realm of robotics, with specific emphasis on the determination of joint angles, markers are thoughtfully positioned on the robot's joints or limbs. As the robot undertakes its movements, cameras vigilantly track these markers, while sophisticated algorithms are deployed to meticulously calculate the intricate spatial relationships between them. These relationships, in turn, furnish invaluable insights into the robot's posture, orientation, and the underlying patterns of its motion. By scrutinizing the dynamic changes in the positions and orientations of these markers, one can astutely infer the robot's joint angles. This capability endows marker-based systems with an inherent value, particularly in applications where precise control over robotic arms or the locomotion of humanoid robots is imperative.

It is imperative to acknowledge that while the fundamental concept of markers may appear straightforward, the underlying algorithms tasked with the interpretation of captured data can be intricately complex. These algorithms are tasked with managing an array of challenges, including instances of occlusion, where

one marker may obscure another, and the impact of fluctuating lighting conditions on marker visibility.

In summation, marker-based pose estimation occupies a pivotal niche within the realm of robotics, proffering a robust and often user-friendly avenue for ascertaining object pose. By meticulously scrutinizing the relative positions and orientations of markers, particularly within the context of robotic joints, we are endowed with the capacity to glean nuanced insights into robot kinematics, thereby facilitating more refined and precise control over robotic entities.

2.2.1 Marker Systems & Calibration Techniques

Pose estimation stands as an indispensable pillar in both the realms of robotics and computer vision, serving as the bedrock upon which systems discern their spatial orientation and relative position within their given environment. Within this context, marker systems emerge as a linchpin, orchestrating the attainment of precise pose estimations, with their calibration techniques serving as the vanguard in safeguarding accuracy and reliability in the pose estimation process.

The paper referenced in [6] prominently accentuates the realm of marker-based methodologies. A salient focal point of this document lies in underscoring the paramount significance of calibration, a fact that assumes even greater prominence when multiple cameras are in play. The calibration process assumes the pivotal responsibility of ensuring that the pose estimations derived from the markers remain consistently accurate across all cameras within the system. The outcomes of this calibration process, exemplified visually in *Figure 6*, not only

illuminate the calibration's procedural intricacies but also showcase its far-reaching consequences.

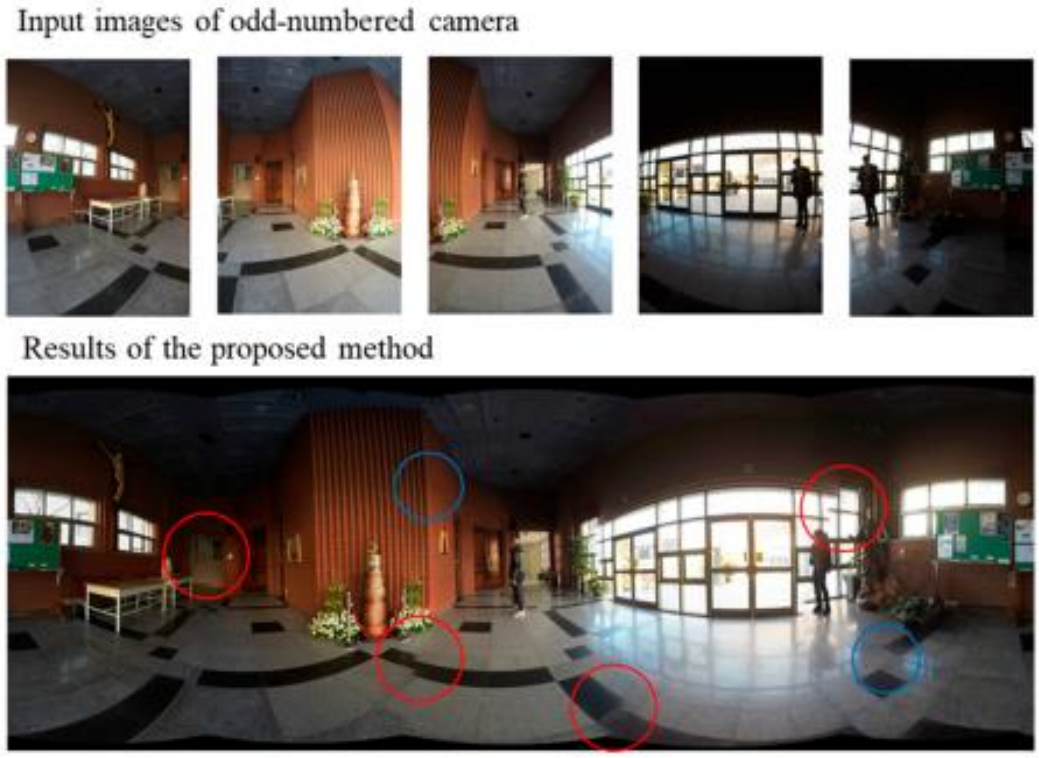

Figure 6:Results of accuracy due to camera calibration [6].

In [7] Marker-based pose estimation utilizes identifiable markers on instruments to deduce their position and orientation from captured images. While promising for endoscopic instrument control, its accuracy is influenced by several factors. Occlusions can obscure markers, leading to inaccuracies. Shadows, reflections, and lighting variations might interfere with consistent marker detection, potentially causing pose estimation errors. Calibration is crucial; inaccuracies in determining the camera's parameters can result in systematic errors. The geometric arrangement of markers is also pivotal; closely placed or non-optimally configured markers might limit accurate pose deduction, especially during rotations.

Rapid instrument movements can challenge consistent marker tracking, introducing transient errors. The method's accuracy was evaluated against an X-ray imager's ground truth, with RMS errors indicating its precision. However, these errors can vary based on the challenges mentioned. In essence, while marker-based estimation offers structured pose determination, its efficacy is contingent on addressing these inherent challenges. This accuracy could be observed in *Figure 7*.

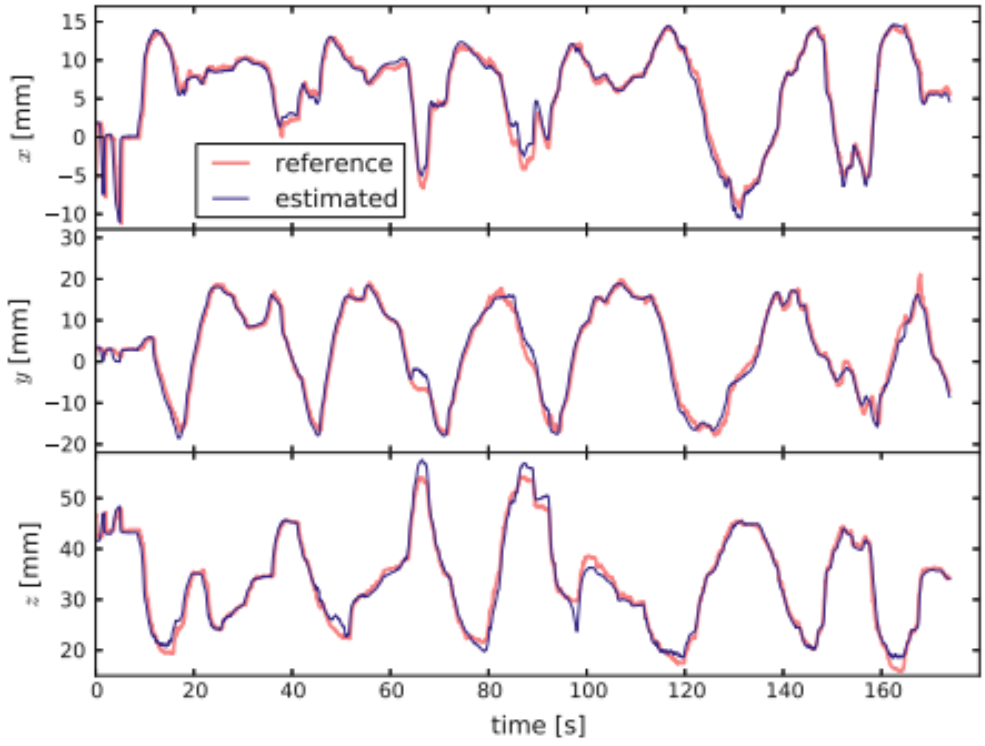

Figure 7: Accuracy of pose estimation of a marker-based system [7].

Lastly, [8] unfolds as an intellectual tour de force, proffering an exhaustive analysis of fiducial markers. These markers, provided with known size and shape, often sport distinctive black and white patterns, although some iterations incorporate a spectrum of colours. The document astutely observes that while coloured markers wield certain advantages, such as expedited detection times,

their resilience dwindles in the face of increased distances and varying angles. The narrative further introduces the STag, a remarkable fiducial marker package that lays emphasis on the robust stability of pose estimation measurements. Distinguished by a unique circular pattern at their core, these STag markers assume centre stage in refining the initial homography through elliptical fitting. The fruits of their extensive empirical endeavours, unveiled in *Figure 8,* serve as a visual testament to the markers employed in their research voyage, featuring the likes of ARTag, AprilTag, ArUco, and STag, each with its distinct imprint on the realm of pose estimation.

Camera	*Angle*	*Light*	ARTag			AprilTag			ArUco			STag		
			mean	std	rate	mean	std	rate	mean	std	rate	mean	std	rate
Logitech	0°	normal	2.831	0.158	45.833	2.450	0.068	99.539	**2.386**	0.176	**99.552**	3.317	0.347	92.825
		shadow	3.132	0.102	45.045	3.417	0.039	**100.000**	**2.012**	0.053	99.545	3.380	0.092	94.444
	10°	normal	**0.625**	**0.047**	46.083	0.884	0.168	**99.552**	1.773	0.115	99.548	1.118	0.135	91.855
		shadow	2.039	**0.081**	45.455	**1.386**	0.136	**99.541**	1.401	0.131	99.539	1.421	0.329	93.953
	20°	normal	2.326	**0.055**	46.296	**1.517**	0.080	99.545	2.219	0.178	**100.000**	2.319	0.115	92.793
		shadow	3.129	0.089	45.045	**2.299**	**0.068**	**99.545**	2.744	0.074	99.543	2.971	0.089	93.088
	30°	normal	**1.162**	0.068	46.330	1.325	**0.066**	99.541	1.379	0.122	**100.00**	1.565	0.077	91.818
		shadow	1.527	0.125	45.662	**1.310**	**0.057**	99.537	1.455	0.105	**99.539**	1.728	0.302	94.907
	40°	normal	0.214	0.050	46.083	0.641	**0.024**	99.091	**0.203**	0.025	**99.550**	0.590	0.070	92.991
		shadow	0.564	0.063	45.662	**0.456**	**0.031**	99.087	0.910	0.050	**99.548**	0.605	0.186	96.789
	50°	normal	**0.141**	0.022	44.595	0.365	**0.015**	**99.543**	0.459	0.046	99.087	0.244	0.033	94.064
		shadow	0.156	**0.021**	45.662	**0.022**	0.024	99.528	0.162	0.046	**99.537**	0.047	0.204	95.909
	60°	normal	0.346	0.030	44.395	**0.126**	**0.009**	99.543	0.239	0.011	**100.00**	0.422	0.019	97.727
		shadow	0.473	0.020	45.455	0.264	**0.016**	**100.00**	**0.188**	0.018	**100.00**	0.194	0.558	95.045
	70°	normal	0.738	0.039	45.662	**0.476**	0.044	99.083	0.779	**0.015**	**99.545**	0.699	0.064	99.541
		shadow	0.716	**0.009**	45.872	0.399	0.012	**99.103**	0.607	0.013	99.087	**0.097**	0.282	96.330
	80°	normal	1.270	**0.039**	41.176	**1.052**	0.089	**100.00**	1.238	0.046	99.087	1.258	0.042	98.190
		shadow	1.143	0.042	45.045	0.874	**0.009**	99.552	0.976	0.010	**100.00**	**0.602**	0.235	97.273
Pi camera	0°	normal	1.883	**0.110**	50.249	0.649	0.267	99.010	**0.602**	0.604	99.010	0.627	0.363	**99.502**
		shadow	**0.010**	**0.147**	49.751	0.288	0.611	**99.502**	4.863	0.435	**99.502**	0.411	0.419	**99.502**
	10°	normal	1.532	0.200	49.751	**0.476**	**0.092**	97.887	1.024	0.178	**99.502**	0.571	1.902	**99.502**
		shadow	1.521	0.100	49.451	**0.323**	0.222	**99.502**	4.201	**0.041**	**99.502**	1.801	0.874	**99.502**
	20°	normal	0.062	**0.052**	49.254	**0.028**	0.054	**99.502**	0.138	0.080	**99.502**	0.088	0.168	**99.502**
		shadow	0.487	**0.063**	49.751	0.561	0.152	**99.502**	**0.278**	0.111	**99.502**	0.600	0.184	98.701
	30°	normal	0.256	**0.039**	49.367	**0.035**	0.055	99.502	0.765	0.141	99.502	0.479	0.158	**100.000**
		shadow	0.294	**0.036**	49.751	0.288	**0.036**	**99.502**	0.255	0.067	**99.502**	**0.134**	0.097	**99.502**
	40°	normal	0.092	0.038	49.254	**0.034**	**0.016**	98.658	0.296	0.031	99.502	0.457	0.387	**100.000**
		shadow	0.148	0.034	49.751	**0.026**	0.075	99.005	0.502	**0.016**	**99.502**	0.497	0.025	**99.502**
	50°	normal	0.440	0.059	48.795	0.519	0.025	**99.502**	0.134	**0.018**	**99.502**	**0.080**	0.032	**99.502**
		shadow	0.210	0.018	49.751	**0.025**	0.020	99.502	0.207	**0.016**	99.502	0.434	0.020	**100.000**
	60°	normal	0.163	**0.015**	49.254	**0.022**	0.016	97.887	0.528	0.033	**99.502**	0.026	0.048	**99.502**
		shadow	0.477	0.015	49.412	**0.066**	0.019	**99.502**	0.193	**0.010**	99.005	0.172	0.034	97.279
	70°	normal	**0.131**	0.015	49.751	0.137	0.011	97.902	0.299	**0.007**	**100.000**	0.322	0.048	98.026
		shadow	**0.071**	0.031	49.751	0.175	0.031	**99.502**	0.342	**0.009**	99.005	0.184	0.152	98.990
	80°	normal	0.335	0.019	49.020	**0.235**	0.012	**99.502**	0.518	**0.004**	**99.502**	0.277	0.169	36.634
		shadow	0.169	0.097	49.505	**0.010**	0.025	**100.00**	0.153	**0.016**	99.502	0.257	0.073	94.527

Figure 8: Different angle reading of different fiducial markers [8].

In summation, these works collectively illuminate the profound importance of marker systems and their calibration techniques within the realm of pose estimation. Through meticulous research, experimentation, and visual representation, they chart the course for enhanced precision and accuracy in understanding the spatial orientation and positioning of objects and entities within the vast expanse of robotics and computer vision.

2.2.2 Comparative Review of Kinematic Models in Robotics

Robotics is a multidisciplinary field that integrates mechanical engineering, electronics, computer science, and other domains to design, construct, and operate robots. One of the fundamental aspects of robotics is understanding and modelling the motion of robots, which is achieved through kinematic models. This review aims to provide a comparative analysis of three primary kinematic models: forward kinematics, inverse kinematics, and the Denavit-Hartenberg parameters [9] [10] [11].

Forward kinematics involves determining the position and orientation of the end-effector (or tool point) of a robot given its joint angles and link lengths. It's a direct method where the known variables are the joint parameters, and the desired outcome is the position of the end-effector. The benefits of forward kinematics include its direct computation without the need for iterative methods, making it useful for the simulation and visualization of robot motion. An illustrative example can be found in *Figure 9* which showcases the coordinate frame assignment for a planar manipulator.

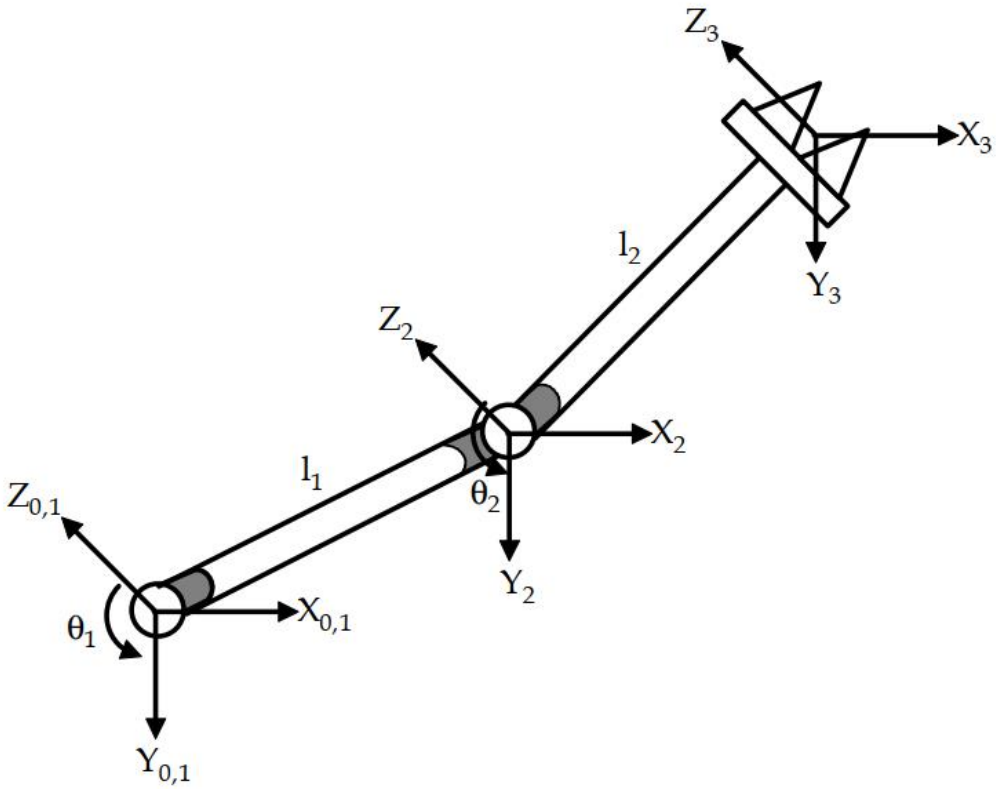

Figure 9: Coordinate frame assignment for a planar manipulator [9].

Inverse kinematics, on the other hand, is the reverse process of forward kinematics. It determines the joint angles required to achieve a desired position and orientation for the end-effector [10]. This method is essential for path planning and control, allowing for precise control of the end-effector's position and orientation. However, it comes with its challenges [9]. There might be multiple solutions for a given end-effector position, and some configurations might not have a solution at all. *Figure 10* presents four positions for a planar manipulator, highlighting the complexities of inverse kinematics.

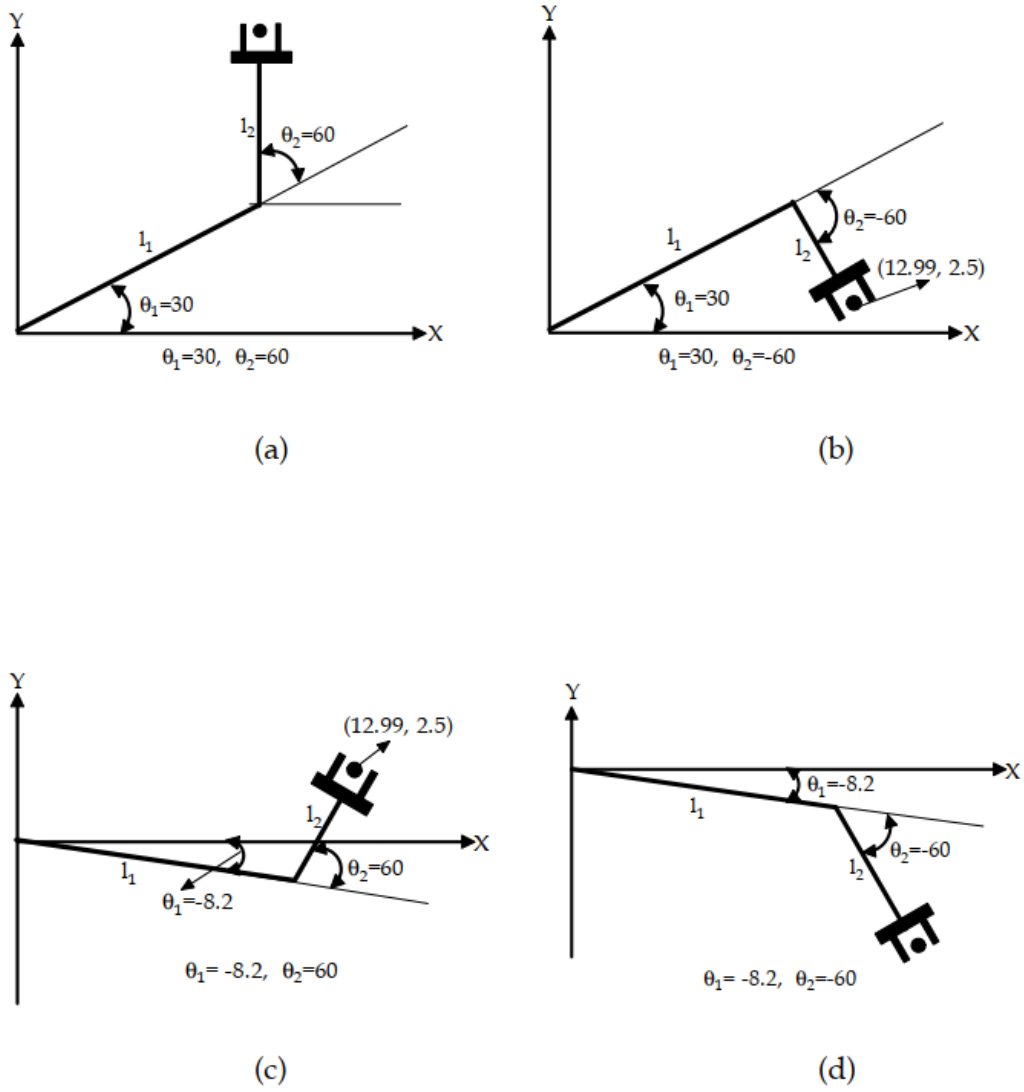

Figure 10: Four positions for a planar manipulator [9].

The Denavit-Hartenberg parameters offer a systematic method to define the geometry of robot manipulators. This approach simplifies the transformation between link frames using only four parameters: link length, link twist, link offset, and joint angle. The benefits of using these parameters include a standardized method for representing robot kinematics and a reduction in the complexity of transformation matrices [11]. An application of the Denavit-Hartenberg parameters can be seen in *Figure 11*, which depicts the coordinate frame and unit line vectors for the Stanford Manipulator.

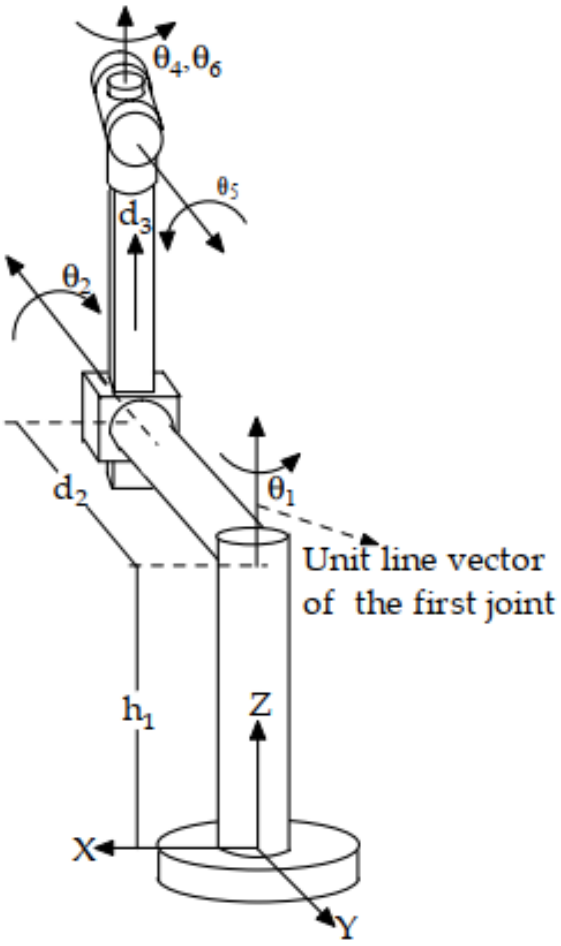

Figure 11: Coordinate frame and unit line vectors [9].

In conclusion, understanding the kinematics of robots is crucial for their design, control, and operation. While forward kinematics provides a direct method to determine the end-effector's position, inverse kinematics offers a way to control the robot's motion based on desired end-effector positions. The Denavit-Hartenberg parameters, on the other hand, provide a standardized method to represent the robot's geometry. Together, these models form the foundation of robot motion analysis and control.

3. Research Methodology

3.1 Analysis of TensorFlow API for Custom Object Detection

In the rapidly evolving domain of object detection, TensorFlow, a deep learning framework pioneered by Google, has carved a significant niche for itself. One of its standout offerings is the TensorFlow Object Detection API, which has been lauded for its proficiency in detecting multiple objects in real-time video streams [12].

What sets this API apart is not just its capability with predefined models. It offers an unparalleled flexibility, allowing users to train it for any custom object. This adaptability ensures that the detection can be tailored to specific needs, making it a versatile tool for a myriad of applications [12].

The open-source nature of the API is one of its primary advantages. This ensures transparency and eliminates any hidden costs that might be associated with its usage [12]. Furthermore, TensorFlow's support for a multitude of platforms, combined with its high-level APIs for model building, makes it a preferred choice for developers and researchers alike [12].

Performance-wise, the API has demonstrated commendable results. It has been benchmarked to showcase performance metrics close to other prominent object detection techniques, such as the Microsoft Azure Cloud [13]. The ability to fine-tune the API to enhance accuracy and expand the range of detectable objects further accentuates its value in the object detection landscape [13].

In conclusion, the TensorFlow Object Detection API is more than just a tool; it's a comprehensive solution for object detection tasks. The myriad benefits it offers, as illustrated in the referenced documents, make it an indispensable asset for those looking to harness the power of deep learning in object detection.

3.2 Construction of a Custom Object Detection Pipeline

In the dynamic world of computer vision, the need for precise and specific object detection models is ever-increasing. Building a custom object detection pipeline via the TensorFlow API requires a meticulous approach that encompasses several key phases which are:

3.2.1 Data Collection:

The process of data collection entailed the meticulous acquisition of pertinent information and samples essential for the execution of the machine learning project. To illustrate, within the context of this study, a comprehensive array of photographs depicting the subject system was meticulously assembled, capturing a spectrum of diverse perspectives and lighting conditions. This approach was adopted with the explicit objective of fostering dataset diversity, a critical consideration within the framework of this research endeavour.

3.2.2 Image Annotation:

Image annotation represents the crucial procedure of ascribing labels or distinctive markers to particular objects or characteristics discernible within the amassed dataset. In this specific instance, the process of manually annotating

the robotic system within each image was meticulously executed. This annotation procedure was facilitated through the utilization of the LabelImg program, which generated XML files meticulously detailing the precise spatial delineation of selected pixels within the images.

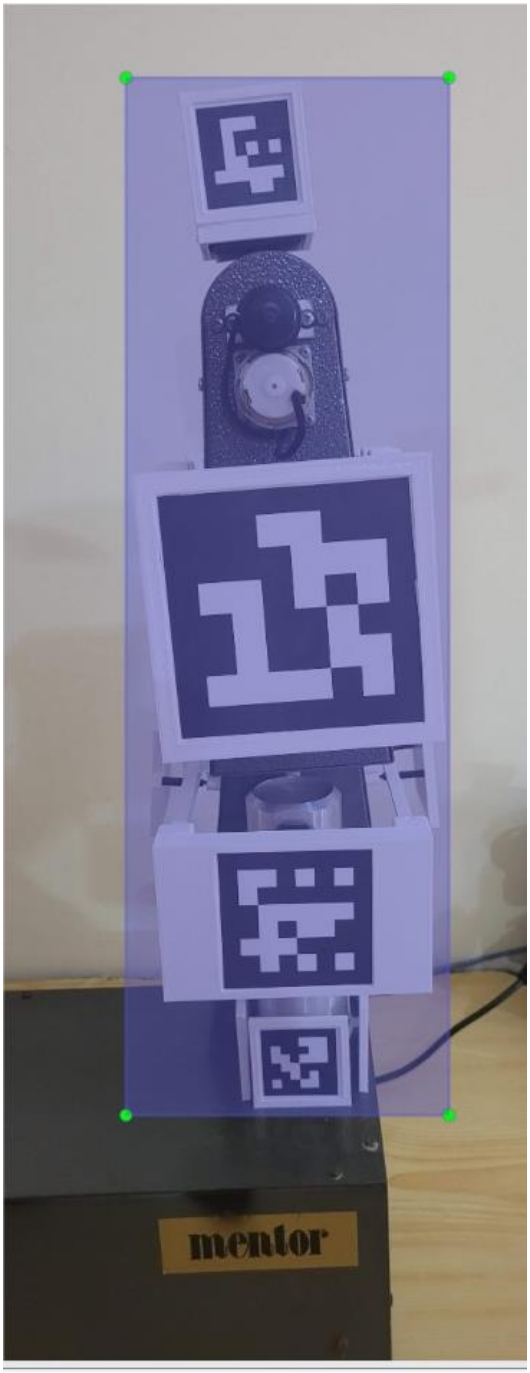

Figure 12: LabelImg software to track the required pixels.

3.2.3 Dataset Split:

Following the meticulous annotation of the dataset, a systematic partitioning into discrete subsets, designated for training, validation, and testing, was executed. This strategic segregation was instrumental in furnishing distinct datasets, each serving a specific purpose in the training, fine-tuning, and evaluation of the machine learning model. The partitioning process was automated through the deployment of a custom script, facilitating the random distribution of data, adhering to a predetermined ratio of 70% allocated for training and 30% designated for testing, thereby ensuring a comprehensive and unbiased evaluation of model performance.

```
[10] #creating two dir for training and testing
     !mkdir test_labels train_labels

     # lists the files inside 'annotations' in a random order (not really random, by their hash value instead)
     # Moves the first 274/1370 labels (20% of the labels) to the testing dir: `test_labels`
     !ls annotations/* | sort -R | head -124 | xargs -I{} mv {} test_labels/


     # Moves the rest of the labels ( 1096 labels ) to the training dir: `train_labels`
     !ls annotations/* | xargs -I{} mv {} train_labels/
```

Figure 13: Randomizing the data split to be utilized for training and testing.

3.2.4 Generate CSV Files:

For the systematic organization and effective management of the annotated data, a structured approach was adopted. Specifically, CSV (Comma-Separated Values) files were generated for this purpose. These CSV files comprehensively encapsulate essential information, including image file names, associated annotations, and pertinent metadata. This systematic data organization strategy was instrumental in streamlining the data preparation process during the

development of the machine learning model, facilitating efficient and structured access to the requisite information throughout the model development phase.

3.2.5 Generate TF Records:

To enhance data compatibility and optimize its suitability for TensorFlow, a conversion process was undertaken, transforming the dataset into the TF Records format. This binary file format is specifically engineered to align with the demands of training deep learning models, offering efficiency gains in the data loading process. This strategic conversion serves to expedite data retrieval and facilitates seamless integration with TensorFlow, thereby contributing to the overall efficiency and effectiveness of the deep learning model training pipeline.

```
#Usage:
#!python generate_tfrecord.py output.csv output_pb.txt /path/to/images output.tfrecords

#For train.record
!python /mydrive/customTF2/generate_tfrecord.py train_labels.csv  label_map.pbtxt images/ train.record

#For test.record
!python /mydrive/customTF2/generate_tfrecord.py test_labels.csv  label_map.pbtxt images/ test.record
```

Figure 14: Utilizing a Python script to generate TF records.

3.2.6 Transfer Learning:

To expedite the process of model development, a strategy of transfer learning was employed, wherein a pre-trained neural network served as the foundational framework. This approach offers the distinct advantage of harnessing the wealth of knowledge acquired by models trained on expansive datasets and adapting it to address the specific requirements of my task. In particular, the ssd_mobilenet_v2_fpnlite_320x320_coco17_tpu-8 model was chosen as the

foundational architecture, with custom adjustments made to its configuration to align it optimally with the project's objectives. This strategic utilization of transfer learning significantly streamlined the model development process, leveraging the collective insights from broader datasets to enhance the performance on the specific task at hand.

```
"""
PIPELINE_CONFIG_PATH=path/to/pipeline.config
MODEL_DIR=path to training checkpoints directory
CHECKPOINT_DIR=${MODEL_DIR}
NUM_TRAIN_STEPS=2000
SAMPLE_1_OF_N_EVAL_EXAMPLES=1

python model_main_tf2.py -- \
  --model_dir=$MODEL_DIR --num_train_steps=$NUM_TRAIN_STEPS \
  --checkpoint_dir=${CHECKPOINT_DIR} \
  --sample_1_of_n_eval_examples=$SAMPLE_1_OF_N_EVAL_EXAMPLES \
  --pipeline_config_path=$PIPELINE_CONFIG_PATH \
  --alsologtostderr
"""
```

Figure 15: Transfer learning parameters set for the training.

3.2.7 Training:

With the dataset meticulously prepared and the model architecture meticulously configured, the commencement of the training phase was initiated. This pivotal stage encompassed the systematic provision of training data to the model, fine-tuning of model parameters, and a series of iterative epochs designed to refine and optimize the performance of the model.

3.2.8 *Model Evaluation & Export:*

Subsequent to the training phase, a comprehensive evaluation of the model's performance was conducted, employing the validation dataset as the benchmark. This evaluation encompassed an assessment of key metrics, including accuracy, precision, recall, and other pertinent criteria, to gauge the model's effectiveness and suitability for the intended task. Upon achieving a satisfactory level of performance, the trained model was meticulously exported, primed for seamless deployment, thereby enabling it to make accurate predictions on previously unseen data with precision and reliability.

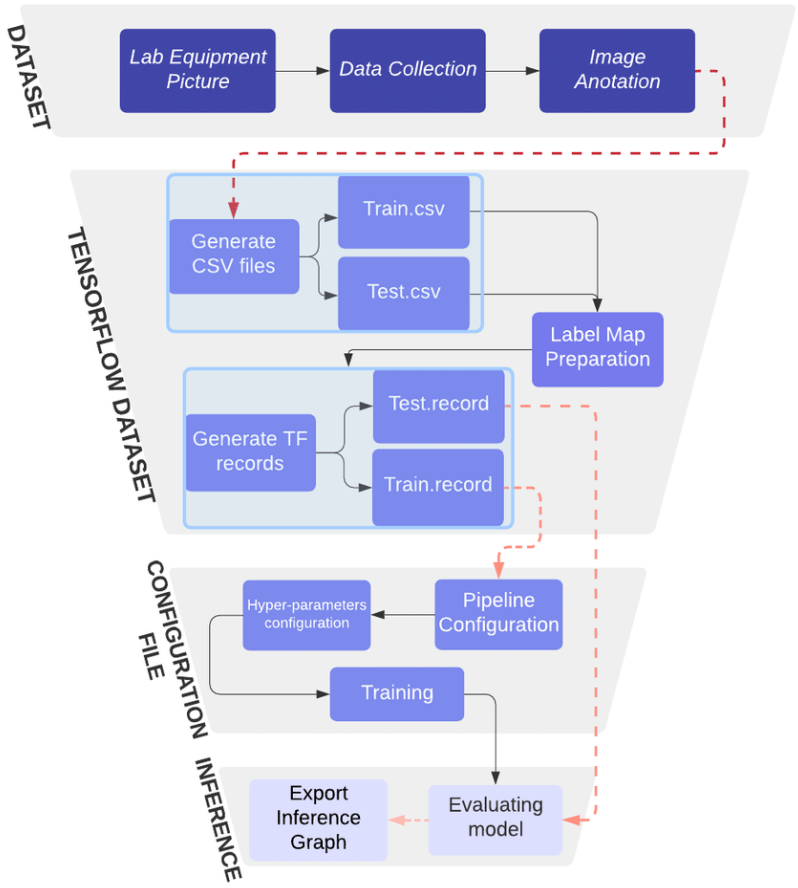

Figure 16: Custom Object Detection Pipeline Example for the forementioned steps [14].

3.3 Marker Selection & Generation

For the successful implementation of pose estimation, the indispensability of robust fiducial markers is underscored. In this regard, the selection of ArUco Markers was grounded in their distinguished attributes, including robustness, accuracy, computational efficiency, and, notably, adaptability. In the initial phase of inquiry, a comprehensive investigation was conducted to ascertain the most suitable matrix size for these markers, as delineated in *Figure 17* below. The exploration encompassed four distinct marker matrix dictionaries, spanning the dimensions of 4x4, 5x5, 6x6, and 7x7. Following meticulous experimentation and a judicious evaluation vis-à-vis the specific requisites of the project, a deliberate preference was established for the 6x6 dictionary.

To facilitate the systematic generation of these markers, a bespoke script was developed, meticulously designed to produce markers of precise dimensions, unique identification codes, and adherence to the selected dictionaries. The scripting solution, characterized by its intrinsic flexibility, not only empowers customization but also facilitates the creation of distinct markers tailored to diverse application contexts and varying environmental requirements. This methodical approach, underpinned by rigorous testing and optimization, underlines the innate robustness and adaptability of the system, thereby substantiating its efficacy across a spectrum of real-world scenarios.

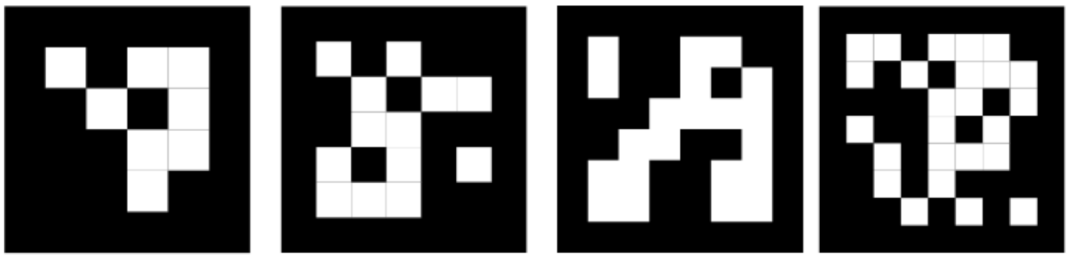

Figure 17: ArUco Markers with different matrix size [15].

3.4 Marker Placement and Design

During the inaugural phase of the design process, a deliberate placement strategy was devised, entailing the affixing of markers on all four sides (front, back, left, and right) of the robotic arm. This particular configuration was meticulously chosen to facilitate comprehensive positional tracking and ensure unfettered orientation visibility for the tracking system, irrespective of the perspective or viewing angle.

The markers themselves were meticulously crafted to seamlessly integrate with the robotic arm's joints, taking into careful consideration the intricacies of shape, size, and mechanical constraints inherent to the arm. An illustrative example of this design is delineated in *Figure 18,* which showcases the CAD blueprint specifically devised to accommodate the markers on the end effector. The tangible outcome of this meticulous design process is exemplified in *Figure 19* which vividly depict the successful integration of markers onto the robotic arm's structure.

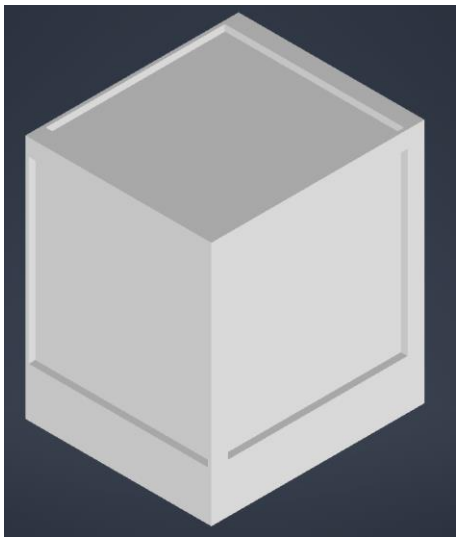

Figure 18: CAD model of end-effector marker placement.

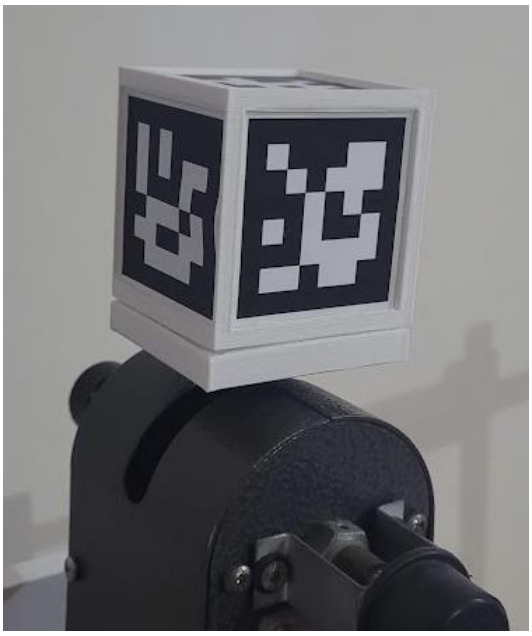

Figure 19: Design implemented on end-effector.

3.5 Camera Calibration

Camera calibration plays a pivotal role in pose estimation by rectifying lens distortion and estimating perspective geometry. In pursuit of this objective, the ChArUco board calibration method was judiciously selected for its efficiency and robustness, combining the merits of ArUco markers with the structure of a

checkerboard pattern. To facilitate this calibration method, a tailored script was designed, serving to generate the requisite ChArUco board configurations, adjusted with precision-guided parameters. Notably, this entailed the careful calibration of marker and ChArUco board dimensions to uphold the desired relative size ratio between them, as the accuracy of calibration is intrinsically tied to this proportion. This calibration parameterization encompassed the determination of square count, square dimensions, marker size, ArUco matrix specifications, and dictionary selection.

The calibration process yielded two distinct ChArUco boards: one of A1 dimensions and another of A3 dimensions, as vividly portrayed in *Figures 20 & 21*. These boards represent essential components in achieving precise and distortion-free camera calibration, thus fortifying the foundation for accurate pose estimation in the system.

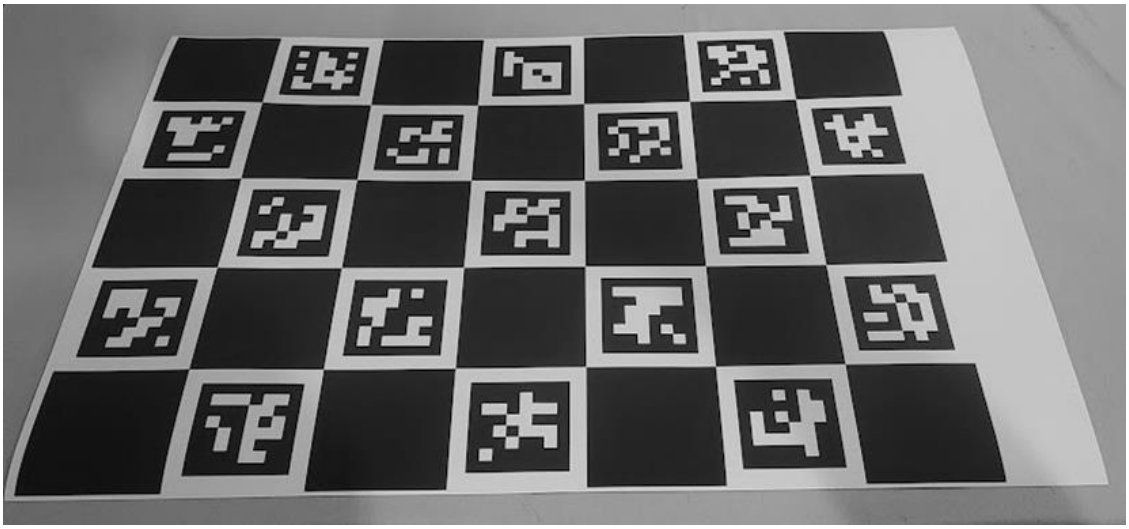

Figure 20: A1 ChArUco Board (5x7)

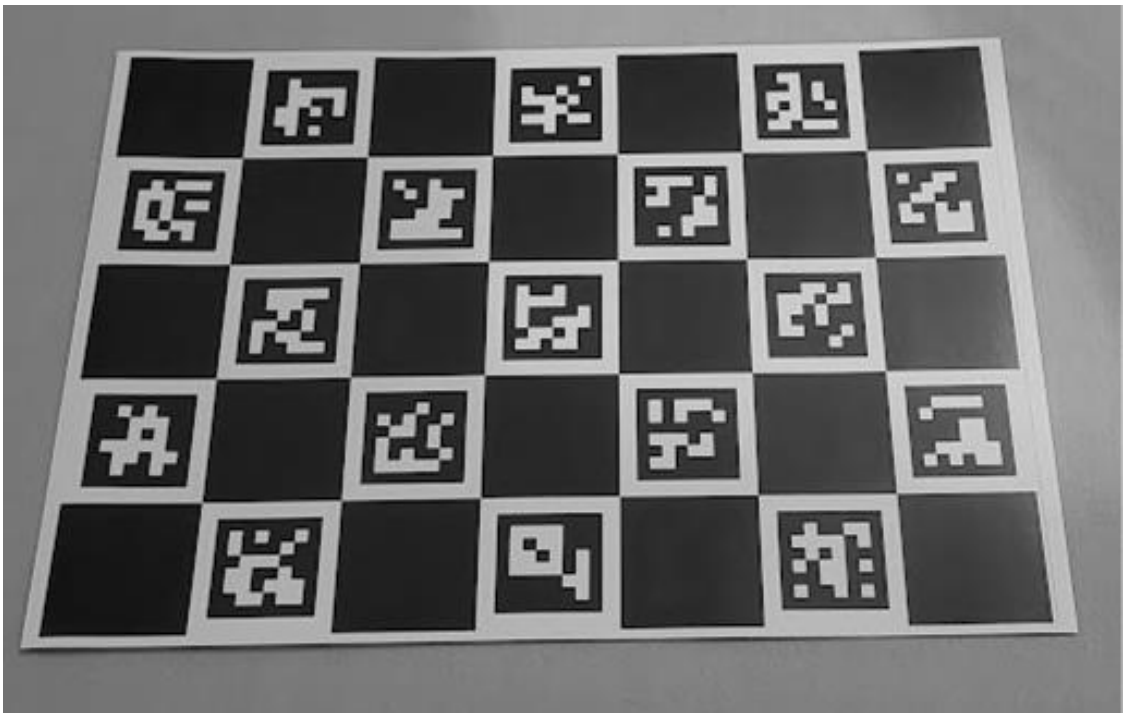

Figure 21: A3 ChArUco Board (5x7)

Numerous photographs were systematically captured of the ChArUco boards using the client's camera, employing varying angles, orientations, and positions, all with the aim to ensure the boards remained consistently within the frame. Subsequently, these diverse images were subjected to a bespoke script, purpose-built for the precise computation of the camera matrix and distortion coefficients. An illustrative representation of this computational outcome is provided in *Figure 22* below. These accurately derived camera parameters assume a pivotal role in facilitating precise image rectification and form a foundational element in the ensuing phases of the pose estimation pipeline.

```
camera_matrix = np.array(
    [
        [6.73172250e02, 0.00000000e00, 3.21652381e02],
        [0.00000000e00, 6.73172250e02, 2.40854103e02],
        [0.00000000e00, 0.00000000e00, 1.00000000e00],
    ]
)

distortion_coefficients = np.array(
    [-2.87888863e-01, 9.67075352e-02, 1.65928771e-03, -5.19671229e-04, -1.30327183e-02]
)
```

Figure 22: Camera matrix and distortion coefficients.

3.5.1 ArUco Marker Data Extraction

Following the successful completion of camera calibration, a distinct script dedicated to the extraction of rotational and translational matrices from the ArUco markers was strategically deployed. This script operates in real-time, processing the live camera feed, detecting, and decoding the markers, and subsequently computing the requisite rotational and translational matrices. Importantly, these computations are anchored in the camera's perspective, which serves as the reference frame for the world. Furthermore, the camera matrix and distortion coefficients play an integral role in the process, enabling the necessary corrections for the characteristics of the specific camera employed. A visual representation of this dynamic process, along with a display of the calculated rotational matrices, is elucidated in *Figure 23* below.

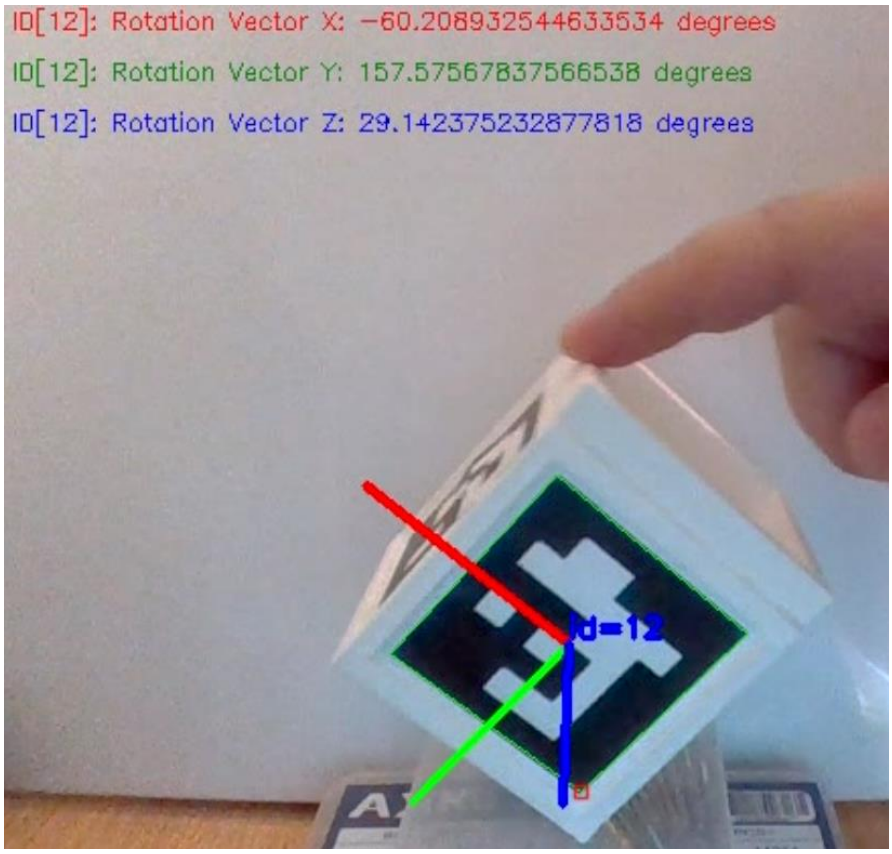

Figure 23: Extracted pose angles.

3.6 CAD Augmentation

Certain adjustments were deemed necessary to address the specific design constraints associated with the robotic system. In particular, the requirement emerged to create a dedicated shaft and mechanical component, as illustrated in *Figures 24 & 25.* The primary objective of this design task was to ensure that, upon attachment, the markers would move in parallel.

Utilizing CAD software, we meticulously delineated the precise specifications, dimensions, and tolerances for the shaft. This rigorous engineering effort was undertaken with a clear focus on ensuring the synchronized movement of the markers. This approach not only guarantees the optimal functionality of the

robotic system but also mitigates the potential for errors or misalignments. The collaborative synergy between CAD tools and robotic systems, exemplified by these two designs, underscores the paramount importance of detailed engineering in achieving precise robotic motion and overall functionality.

Figure 24: 3D Printed Augmenting Shaft

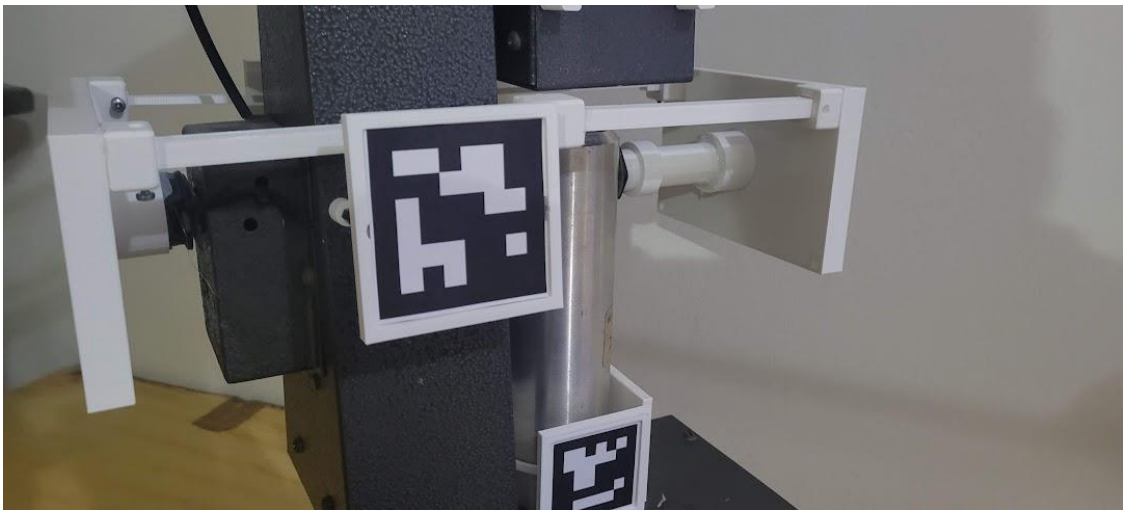

Figure 25 Augmentation on the Robotic System

3.7 Live Virtual Prototype using DH Parameters in MATLAB

The creation of this Live Virtual Prototype using Denavit-Hartenberg (DH) parameters in MATLAB serves the fundamental purpose of establishing an interactive simulation environment for a robot arm. The software framework is engineered to initiate a TCP/IP connection on the localhost, thereby enabling the reception of real-time data. This data encapsulates various joint angles relevant to the robot's configuration.

Upon reception, the software parses this incoming data string, extracting crucial information regarding the positions of distinct components of the robotic arm, including the base, shoulder, elbow, end effector, and the pitch of the end effector. Subsequently, the system employs this parsed data to dynamically plot and visualize the current configuration of the robot in real-time. This visual representation offers a live portrayal of the robot's movement.

The structural and kinematic attributes of the robot are meticulously defined through the utilization of DH parameters. These parameters serve as a concise and efficient means of characterizing the robot's geometric and joint configurations. This methodological approach greatly expedites the simulation and visualization of complex robotic systems.

In the event of parsing errors, the robot maintains its default position, ensuring the safety and stability of the simulation environment. Overall, this live virtual prototype serves as an invaluable tool for testing, visualization, and

comprehension of the robot's motion and behaviour, providing critical insights prior to its implementation in real-world scenarios.

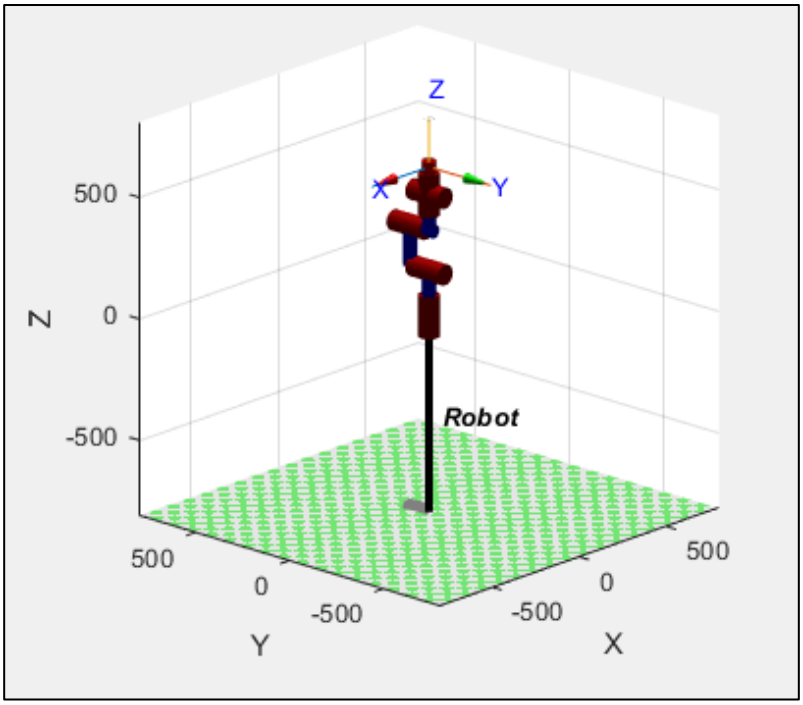

Figure 26 Live Virtual Prototype

3.8 Block Diagram of The Overall System

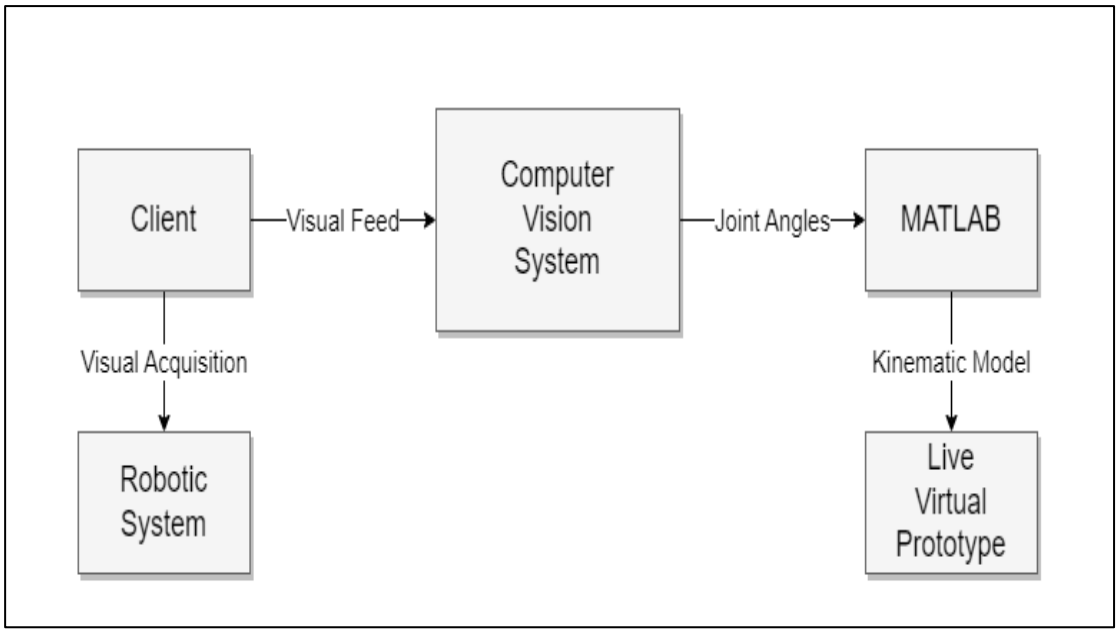

Figure 27: Block Diagram of The Overall System

3.9 Flowcharts

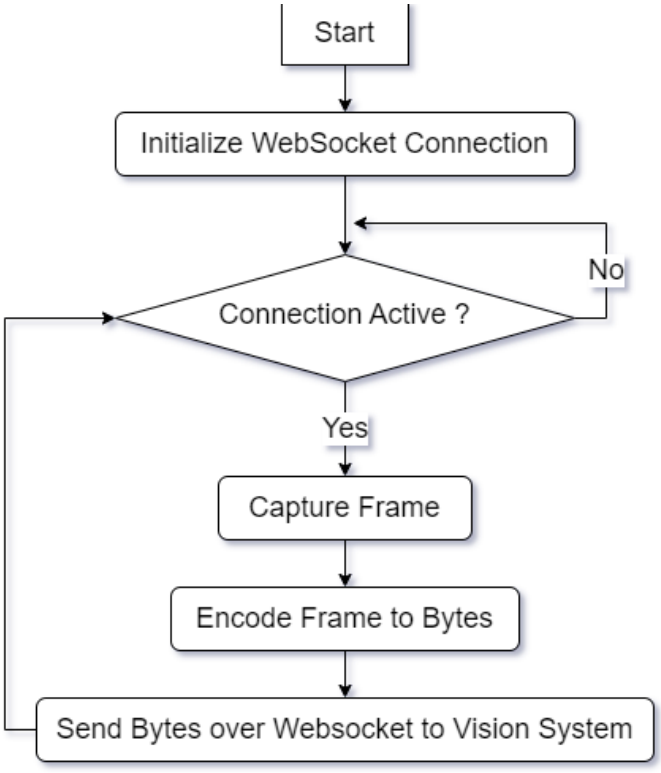

Figure 28: Client System Flowchart

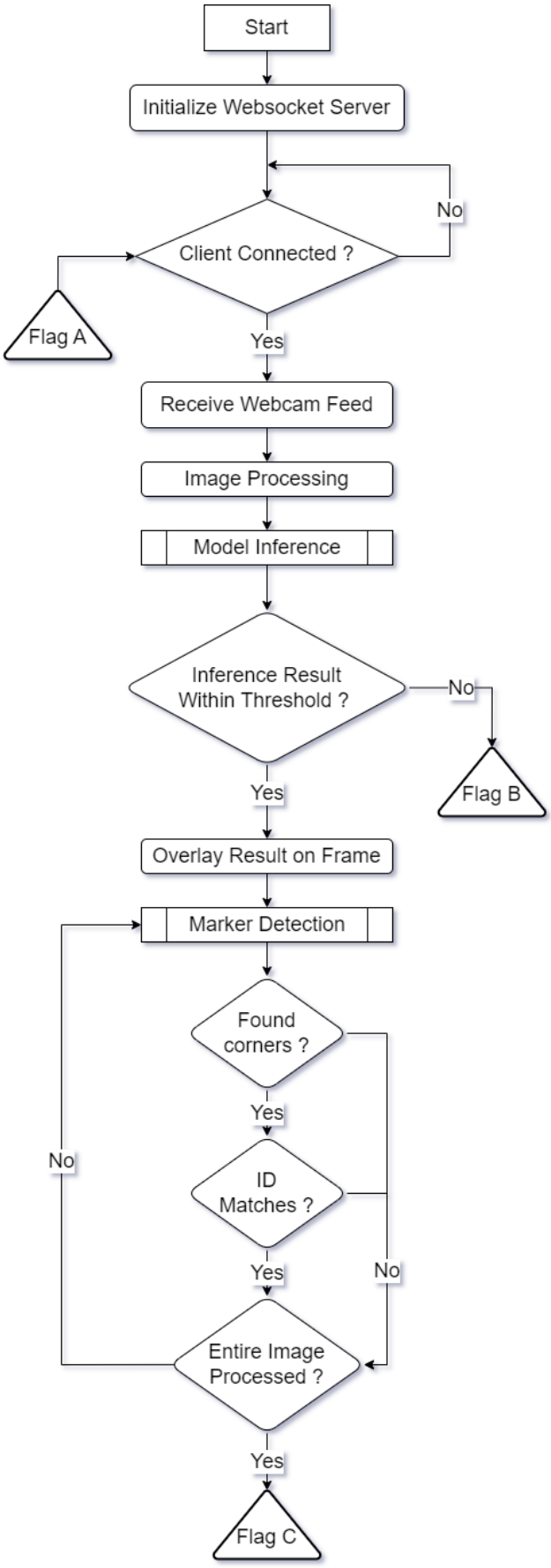

Figure 29: Vision System Flowchart Part 1

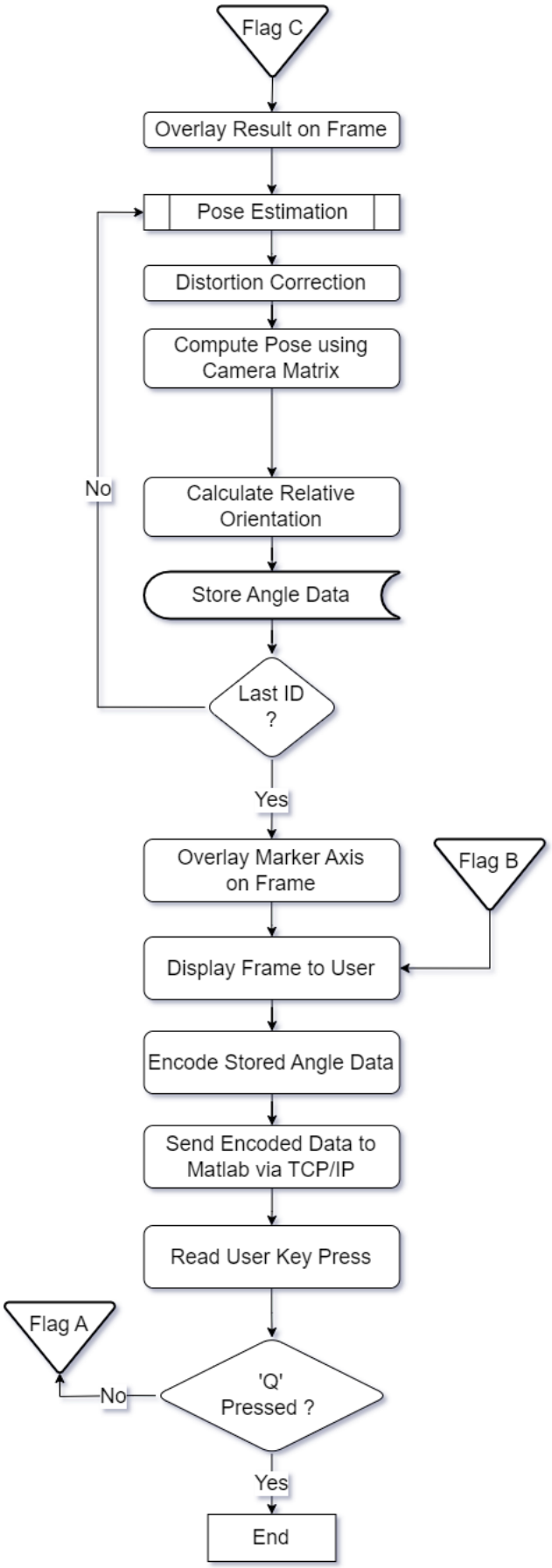

Figure 30: Vision System Flowchart Part 2

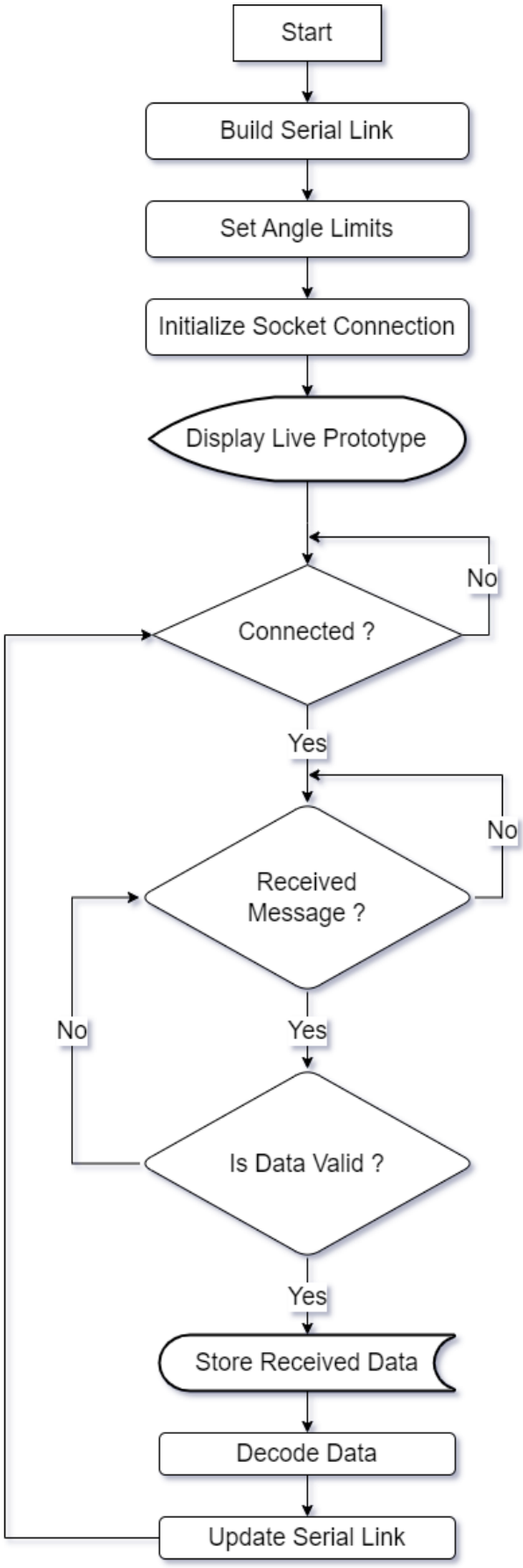

Figure 31: MATLAB System Flowchart

4. Analysis of Results and Discussion

4.1 Initial Testing

During the course of this study, an extensive battery of tests was conducted with precision to assess the implementation of the TensorFlow-based tracking system, with a specific focus on the TensorFlow Lite iteration. The primary aim of these meticulous assessments was to ascertain the practical viability of the model across a wide spectrum of hardware environments.

In the initial phase of experimentation, high-end Raspberry Pi systems were chosen as the testbed, and the TensorFlow Lite framework was thoughtfully employed for the evaluation. TensorFlow Lite was selected due to its reputation for being less resource-intensive, suggesting suitability for settings characterized by limited computational resources. However, the outcomes stemming from this initial phase were, unfortunately, far from satisfactory. Despite the Raspberry Pi's standing as the most potent model within its category, it grappled to align with the performance standards established for it. The constraint became conspicuous, and it was clear that TensorFlow Lite, while optimized for devices of modest computational capacity, was unable to fulfil the stringent demands of this particular environment.

In light of these discerned limitations, a strategic shift in approach became imperative as efforts were made to unearth avenues for performance enhancement. The overarching hypothesis postulated that a Windows-based workstation, endowed with significantly superior processing capabilities, could adeptly handle the requisite computational tasks. Anchored by the promise of its

formidable hardware specifications, it was anticipated that the Windows workstation would surpass the Raspberry Pi's capabilities by a considerable margin. To empirically test this hypothesis, an ingenious pipeline was devised to seamlessly facilitate the transmission of the client's webcam feed directly to the Windows workstation for processing.

Substantiating these anticipations, the results indeed unveiled a realm of promise. The Windows workstation showcased a substantial boost in performance compared to the Raspberry Pi, effectively validating the hypothesis. Nevertheless, this strategic pivot introduced its own set of challenges, primarily stemming from network-related constraints.

Upon deployment of the system within a hotspot environment, discernible performance degradation was observed. This outcome was, perhaps, inevitable, given that hotspot connections are inherently susceptible to the vicissitudes of network instability, often failing to provide the requisite stability and high-speed data transfer necessary for computationally intensive tasks. Nonetheless, it is noteworthy that, even when grappling with these limitations inherent to hotspot usage, the results still managed to outperform those obtained through the Raspberry Pi setup. This offers a glimmer of hope and signals progress in terms of system performance and, importantly, feasibility within diverse environments.

4.2 Tracking Testing

In the comprehensive testing phase dedicated to tracking the robotic system, a series of meticulously designed experiments were conducted across diverse environmental settings. These experiments sought to evaluate the model's performance under varying conditions, including scenarios marked by occlusions and differing distances between the observer and the robot.

4.2.1 Tracking ArUco markers

A script was implemented to facilitate controlled rotations of the cube, which featured ArUco markers affixed to its surfaces. This script served the purpose of systematically observing the tracking performance along all axes during the cube's motion. Notably, the script was designed to accommodate the detection of multiple markers concurrently.

The results, as depicted in *Figures 32 - 34*, demonstrate the model's ability to successfully detect multiple markers from varying angles. These findings emphasize the model's competence and adaptability in addressing diverse tracking scenarios.

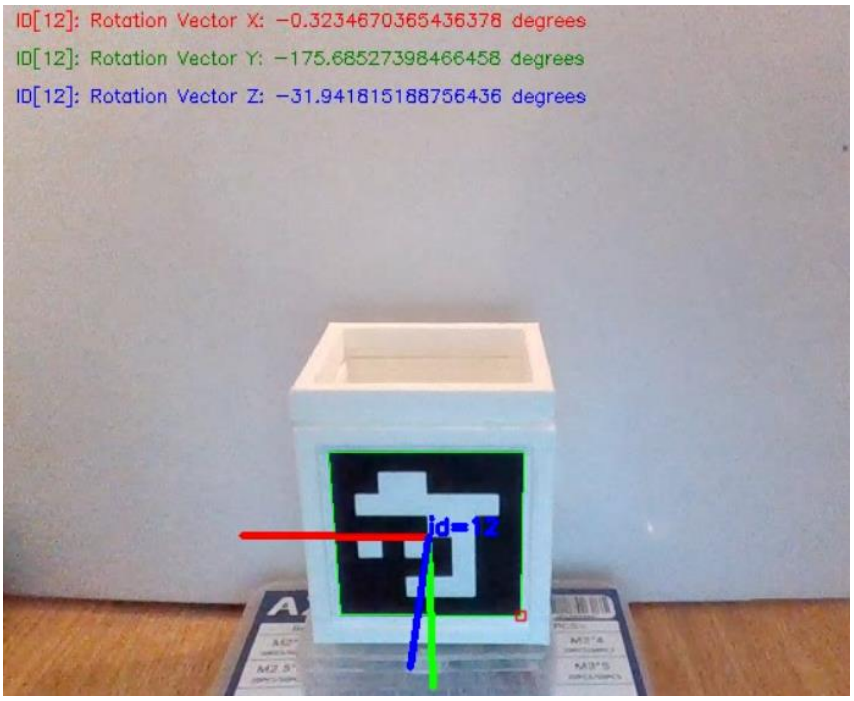

Figure 32: ArUco marker Vector X set at 0°.

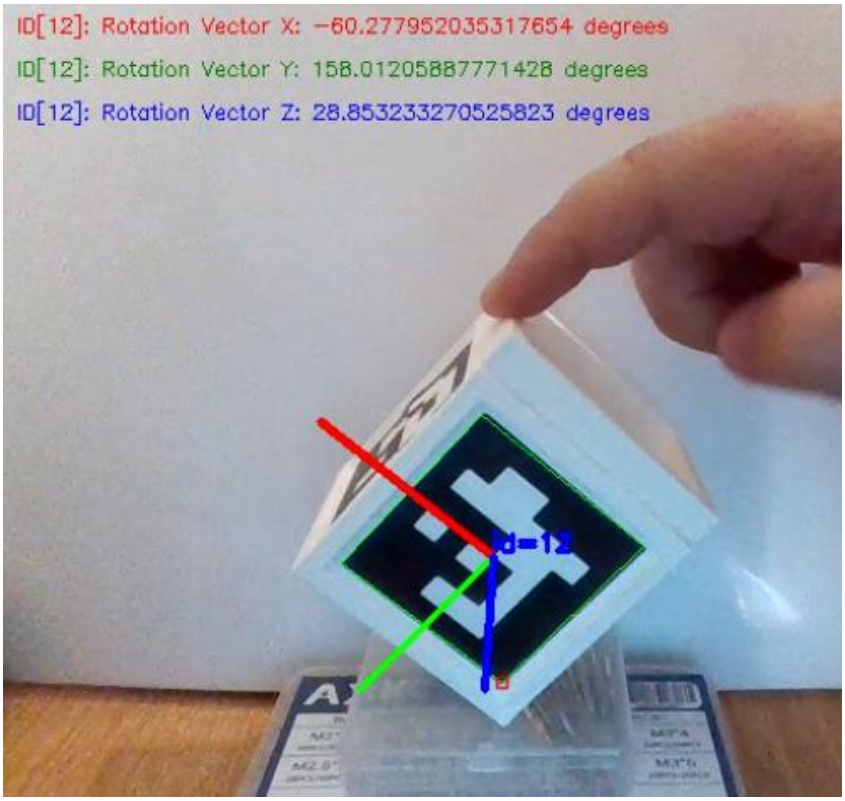

Figure 33: ArUco marker Vector X set at 60°.

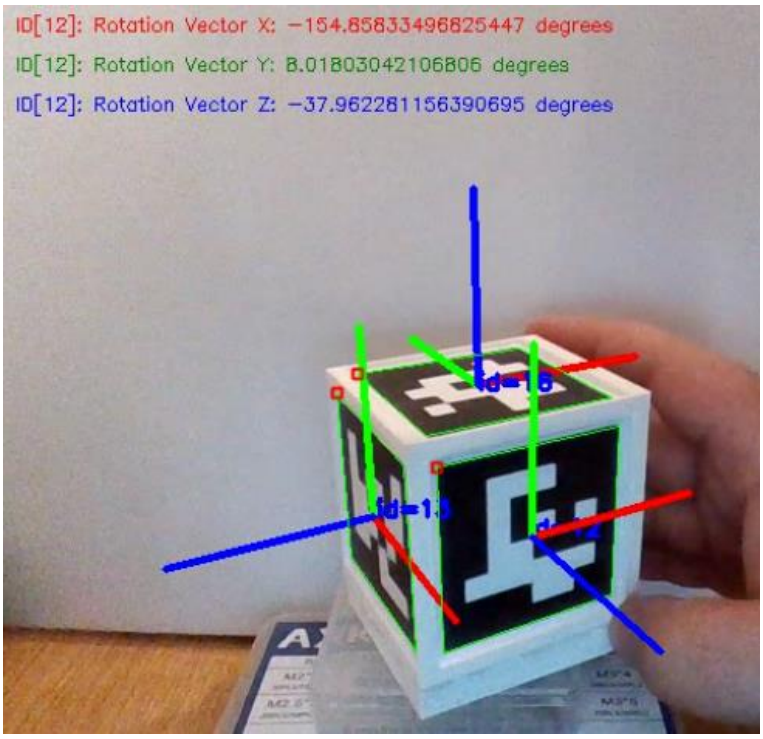

Figure 34: Detection of multiple ArUco markers with change in all Vectors.

4.2.2 Tracking of Robotic Arm

The main focus was to assess the model's reliability in accurately tracking the robot's movements and pose, even when faced with challenging environmental factors. This testing involved tracking the robotic arm and with an obstruction, simulating real scenarios. This could be observed in *Figures 35 & 36*.

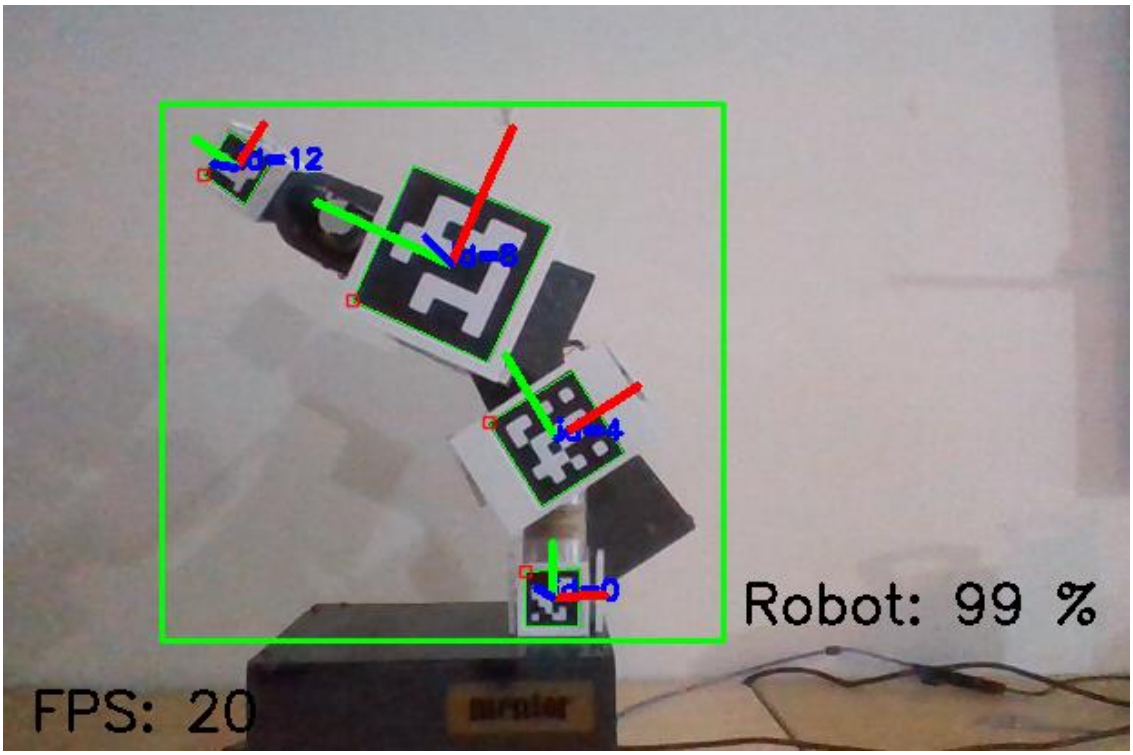

Figure 35: Tracking the robotic arm unobstructed.

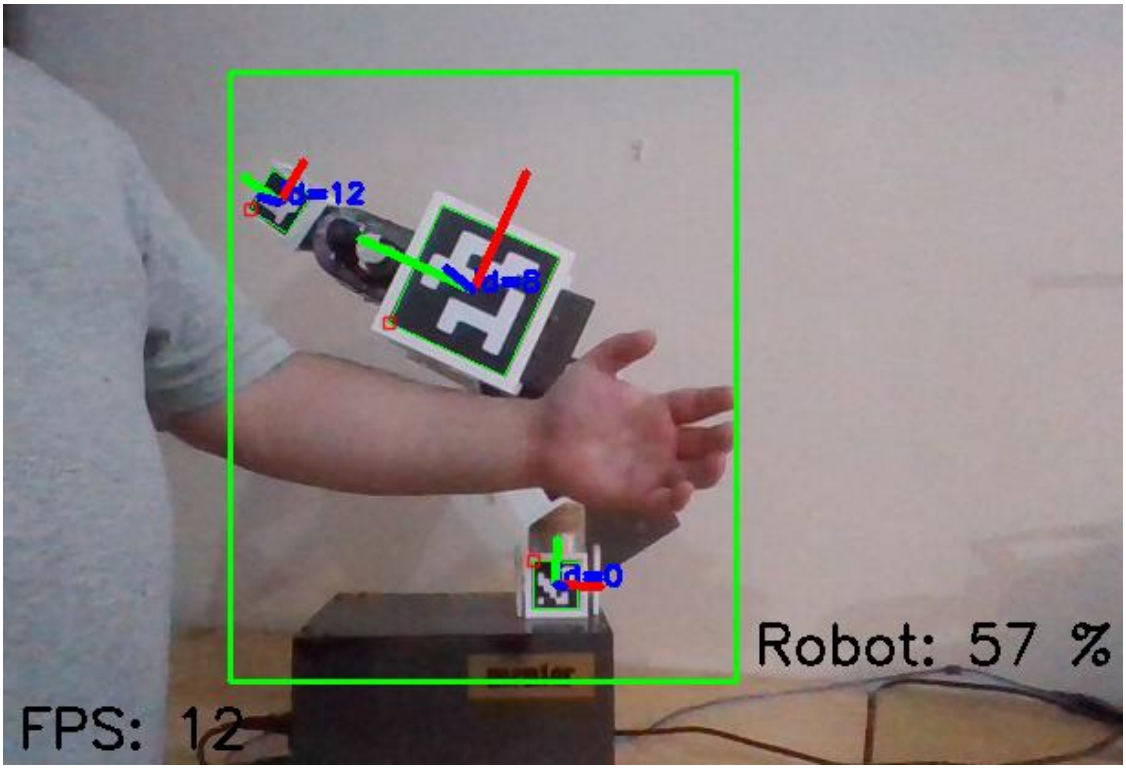

Figure 36: Tracking robotic arm obstructed.

The results unveiled a consistently impressive performance, with the model maintaining a reliability rate exceeding 80% across the majority of the trials for the unobstructed.

However, a noteworthy observation emerged when the robot ventured into regions where its visibility to the tracking system was compromised due to occlusions. In such instances, the model's performance exhibited a discernible decline, resulting in a reduction in the reliability percentage. This phenomenon was anticipated, as occlusions inherently obstruct the model's ability to maintain continuous tracking.

Notably, as the distance between the observer and the robot increased, further tests unveiled a reduction in tracking reliability. This phenomenon can be attributed to the inherent challenges posed by greater distances, which can diminish the quality and clarity of the visual data available to the tracking system.

4.2.3 Analysis of tracking model

Through careful monitoring of the model's performance on *TensorBoard*, a collection of informative plots has been compiled. These plots provide a comprehensive view of the machine learning process, showcasing key metrics such as classification loss, localization loss, regularization loss, and the total loss, which combines these components.

These visual representations utilize a standard Cartesian coordinate system, where the x-axis tracks the progress of epochs, and the y-axis displays the loss values. *Figure 37*, presented below, offers a visual summary of this data.

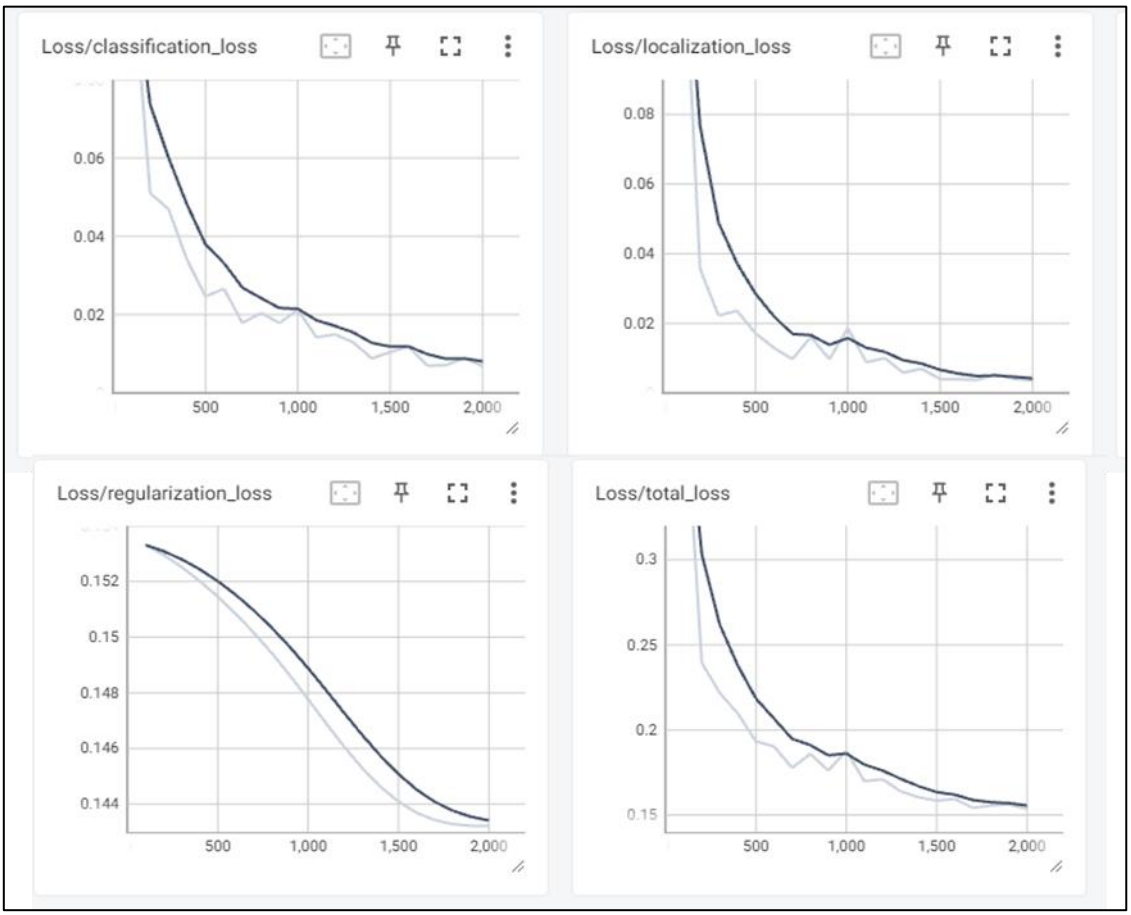

Figure 37: Loss of machine learning model over epochs.

Within this graphical depiction, bright lines represent machine learning loss, while darker lines denote validation losses. A thorough analysis of these results highlights the model's strong performance. The close alignment between validation loss and training loss underscores the model's ability to effectively recognize and learn general patterns, enabling it to make accurate predictions on previously unseen data.

4.3 Pose Estimation

In the domain of Pose Estimation, the objective is to harmonize the previous testing procedures to achieve precise tracking of the robotic system's spatial coordinates and joint angles. The upcoming *Figures 38 & 39* visually depict the outcomes of both the tracking of the robotic arm and the angle of the joints.

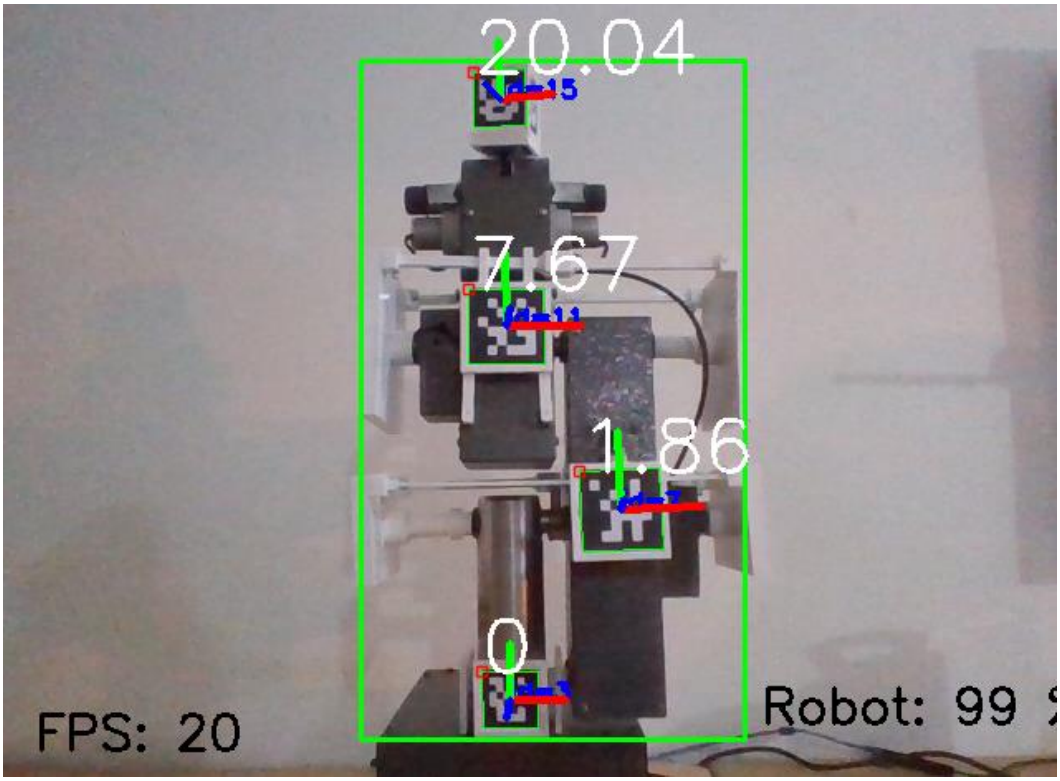

Figure 38: Robot tracking and angle tracking of joints from one side.

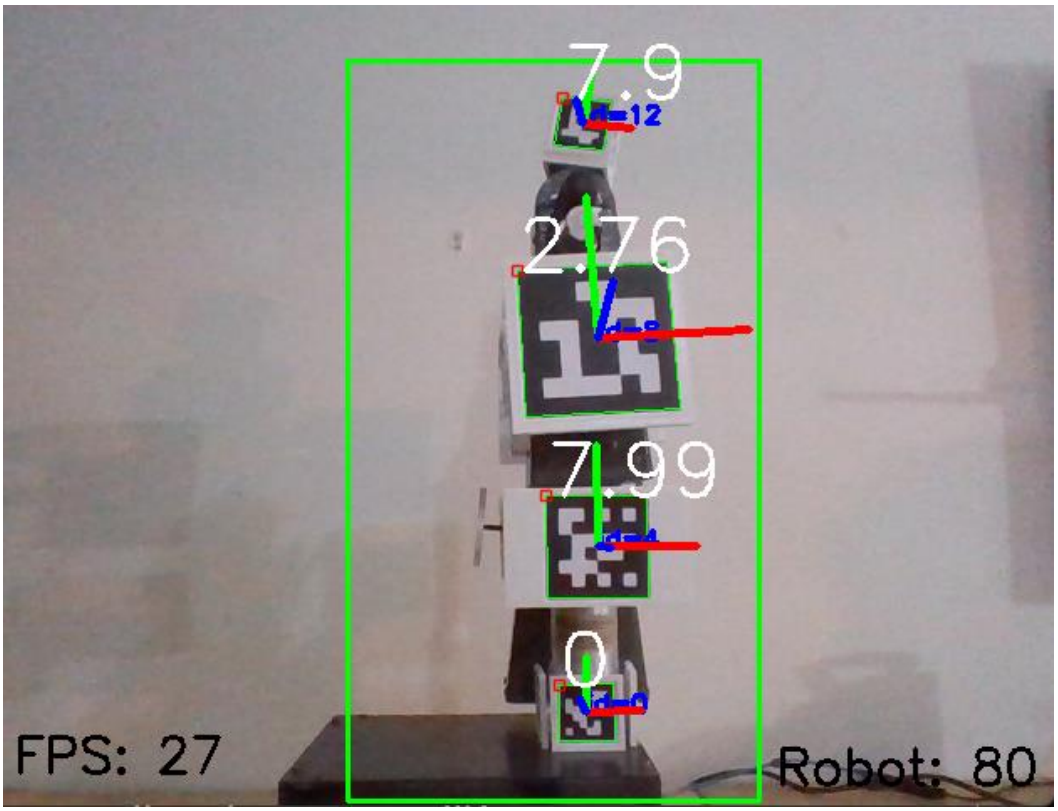

Figure 39: Robot tracking and angle tracking of joints from another side.

4.3.1 Testing Accuracy

In pursuit of precise accuracy assessment for pose estimation, a specialized testing apparatus was meticulously crafted through CAD techniques. The deployment of 3D printing technology played a pivotal role in the construction of this testing apparatus, ensuring precision and repeatability in its design and manufacture.

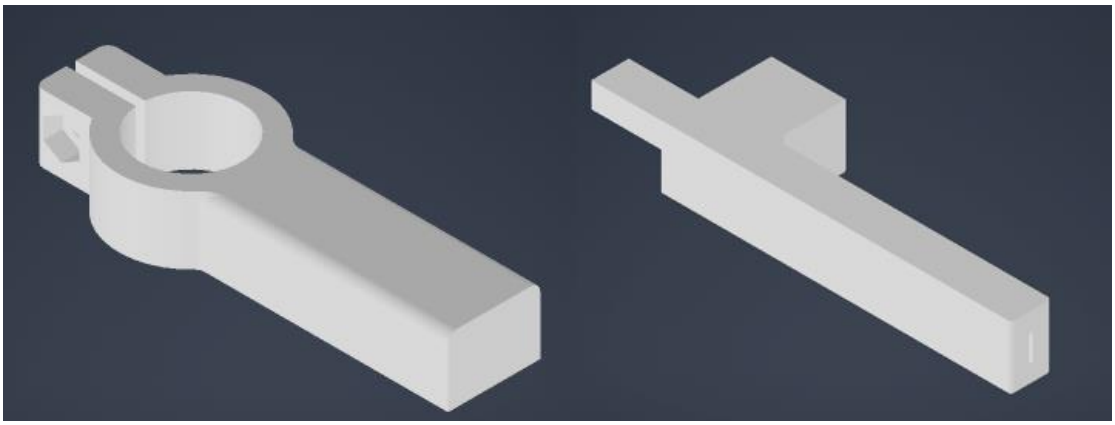

Figure 40: 3D prints used to make the testing apparatus.

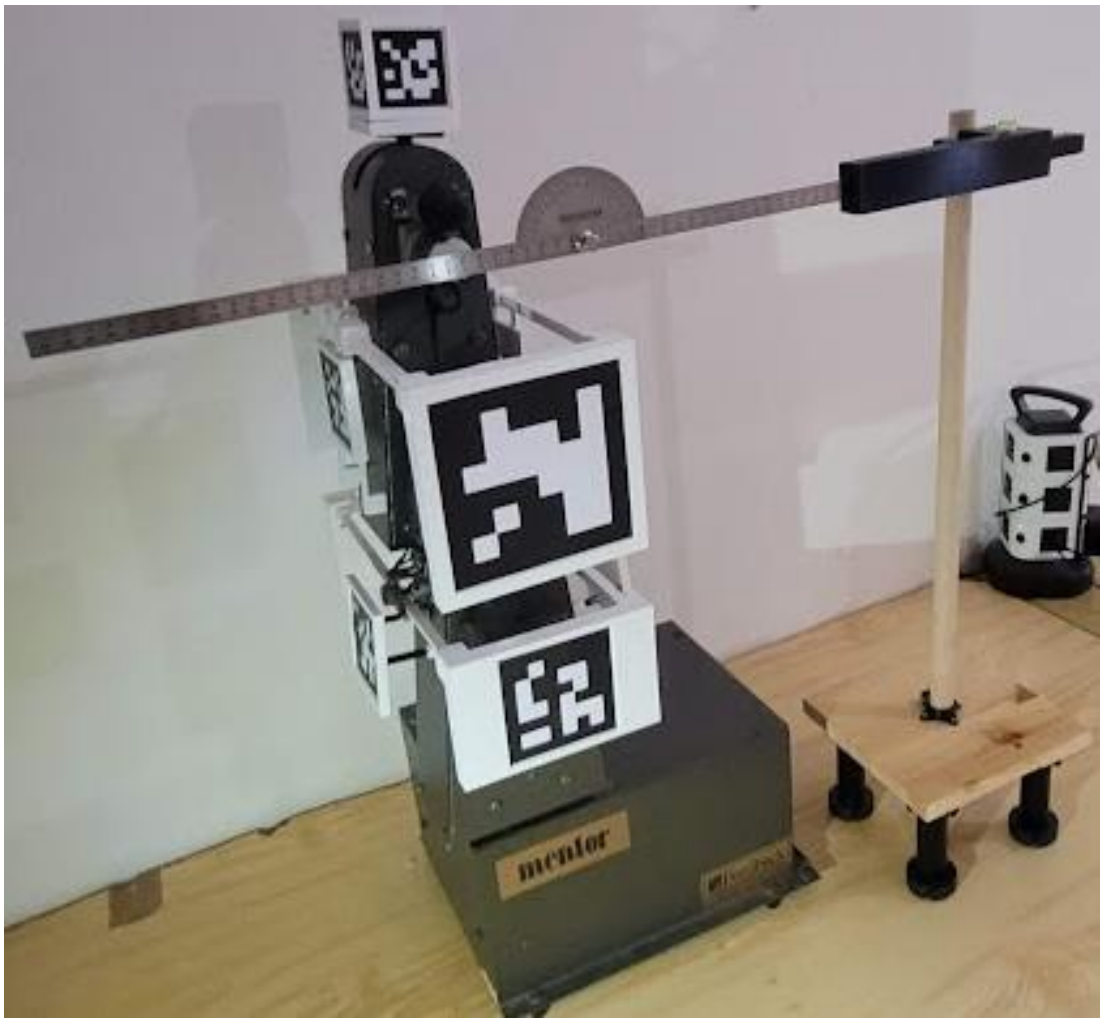

Figure 41: Robotic arm with test setup.

During the testing phase, meticulous attention was given to maintaining optimal levels of precision. To achieve this, a spirit bubble was employed to meticulously level all components of the testing apparatus. This approach was instrumental in ensuring that any discrepancies in measurements were not attributed to misalignment or uneven surfaces but were instead a true reflection of the system's performance.

The primary objective of the testing apparatus was to validate the accuracy of the measured angles. This validation extended beyond a mere comparison with the inputted angles into the robot, as it acknowledged the potential for inherent imprecisions stemming from potentiometer limitations. To establish a robust benchmark for accuracy assessment, a professional-grade protractor was integrated into the testing apparatus.

Figure 42: Application of spirit bubble to keep test setup level.

To further enhance the precision of the apparatus, adjustable feet were incorporated. These adjustable feet provided a means to fine-tune and maintain the

protractor's level, ensuring the highest degree of accuracy during testing procedures.

One of the key functionalities of this testing setup was the ability to discern and record the angle of movement of the marker with remarkable precision. This process facilitated a comprehensive evaluation of the system's pose estimation accuracy, delivering insights into its real-world performance in a controlled and highly accurate testing environment.

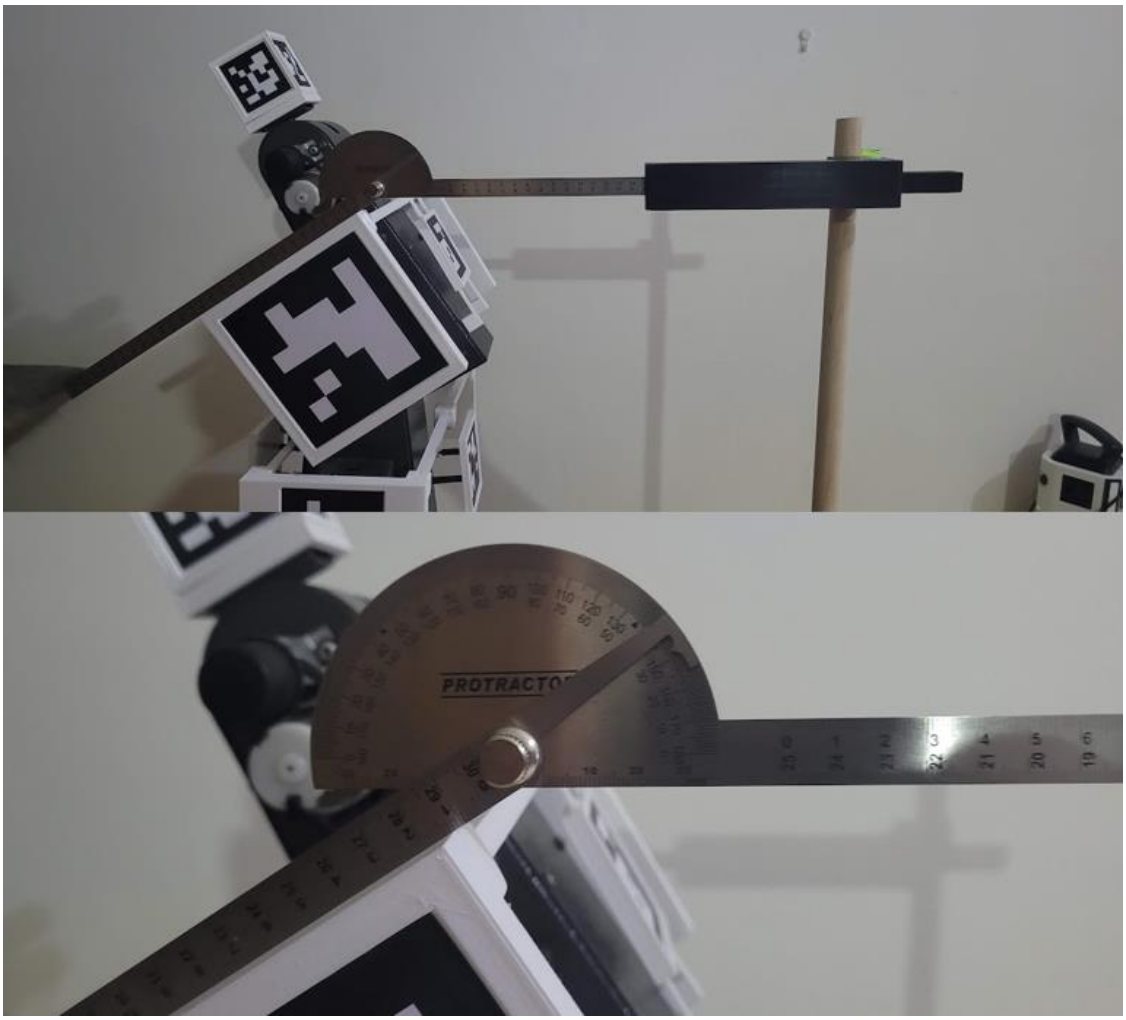

Figure 43: Reading the angle of movement of the elbow marker.

By aggregating the dataset within Microsoft Excel and creating visual plots that put together the sensor angle, protractor-derived angle, and encoder-inputted angle, a lucid depiction of the system's accuracy emerges. The following plots, presented below, provide a concise and coherent portrayal of the interrelationships between these critical variables.

Elbow (Average)									
Sensor	-45	23	8	0	3	0	12	30	48
Protractor	-91	21.8	46	69	2	66	45	17	92
Encoder	-90	-67.5	-45	-22.5	0	22.5	45	67.5	90

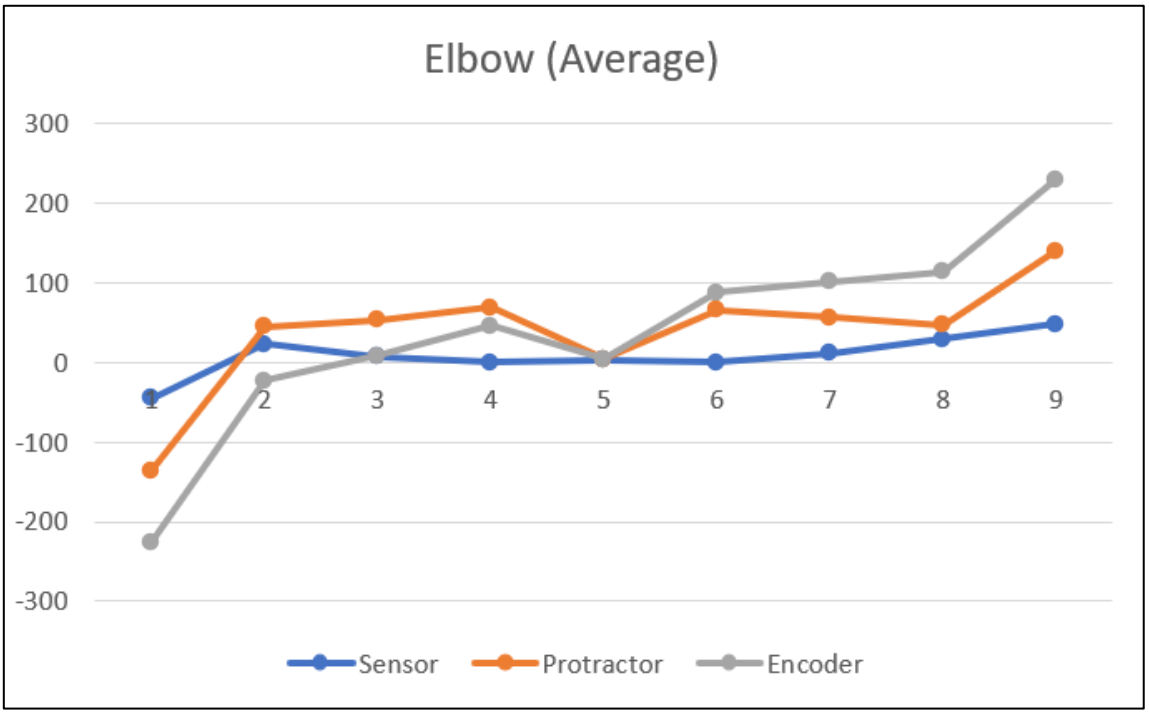

Figure 44: Elbow average from multiple tests.

Shoulder Realistic (Average)									
Sensor	47	23	1	4	14	8	10	31	42
Protractor	0	19.4	36	64	1	68	46	23	90
Encoder	0	22.5	45	67.5	90	112.5	135	157.5	180

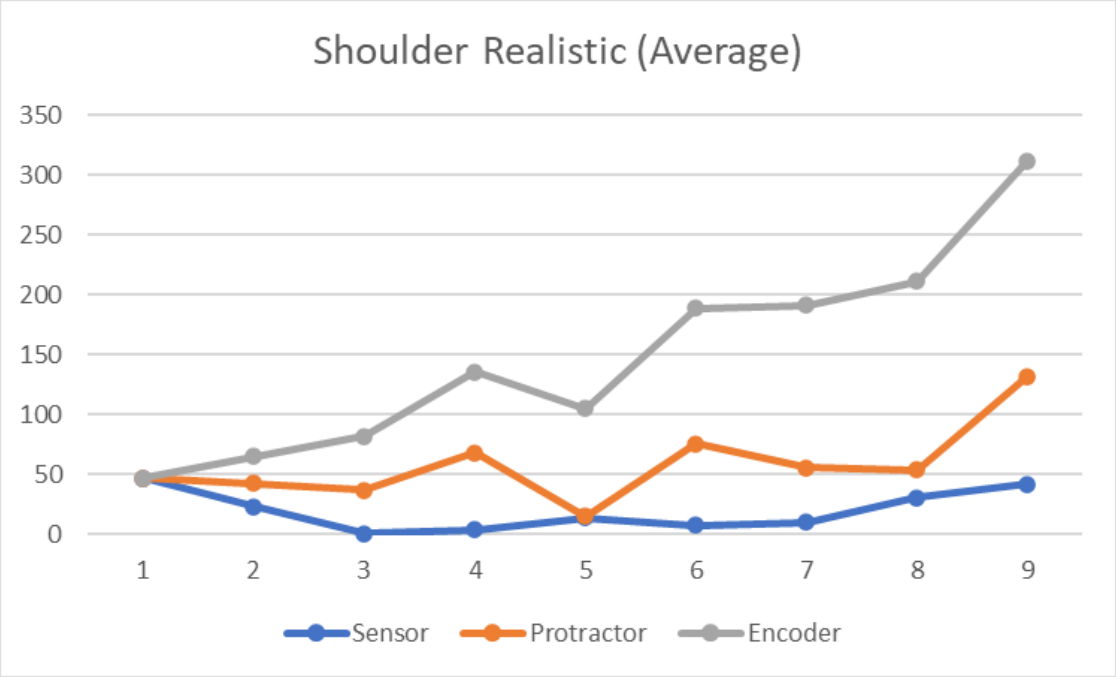

Figure 45: Shoulder average from multiple tests.

As could be noted from the graphs above both the elbow and shoulder show a relationship between the sensor angle, the protractor-derived angle and from the encoder-inputted angle.

4.4 Final Result

By integrating pose estimation, tracking, and the live virtual prototype, a refined outcome will showcase the actual robotic arm with concurrent real-time angle data and a dynamic representation of the arm within the MATLAB environment. These integrated results are visually presented in *Figure 46*.

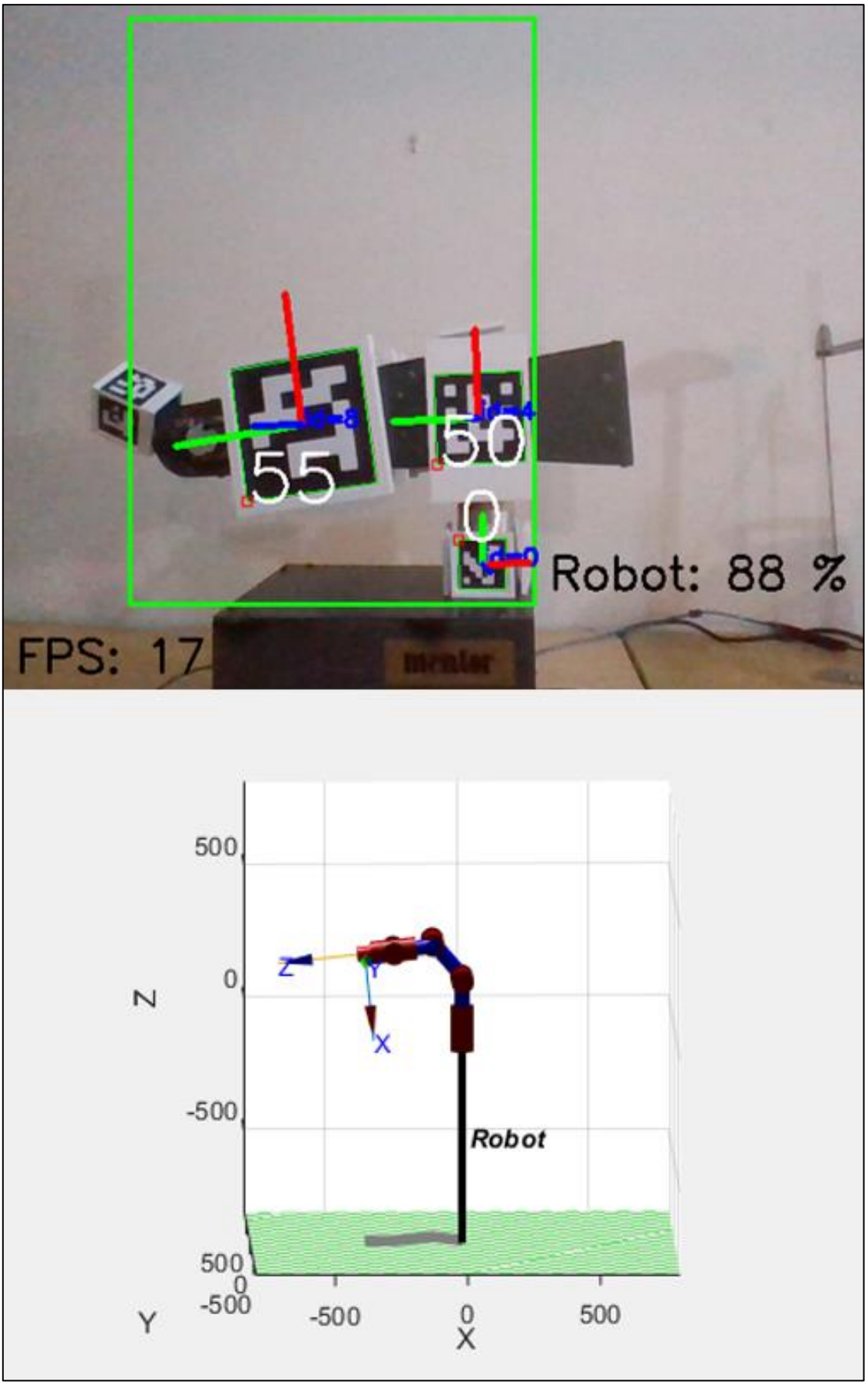

Figure 46: Pose estimation and tracking followed with real-time representation on MATLAB.

5. Conclusions and Recommendations

5.1 Overview

This research journey has been characterized by an unwavering commitment to the refinement and advancement of robotic pose estimation and tracking. It initially set out with the aspiration of utilizing a single-board computer, typified by the Raspberry Pi, as the foundational element of the computer vision system. The objective was to leverage these compact and versatile devices for real-time object tracking and joint angle estimation within the robotic arm. However, the research path was riddled with challenges, revealing formidable limitations in this approach.

The first significant obstacle encountered was the restricted frame rate (fps) of the Raspberry Pi, which severely hindered the system's capacity to swiftly capture and process images. In the realm of real-time tracking, where every millisecond counts, a suboptimal frame rate inevitably leads to imprecise tracking and diminished system performance. Additionally, the demanding computational requirements imposed by advanced object tracking and pose estimation algorithms strained the Raspberry Pi's processing capabilities, resulting in operational inefficiencies.

Faced with these formidable challenges, the research trajectory underwent a transformative shift, departing from reliance on the Raspberry Pi to embrace a more robust and powerful computing platform. The research pivoted toward a Windows-based machine, fortified with TensorFlow and dedicated accelerator cards. This transition marked a critical turning point, mitigating the constraints of

frame rate and processing power while opening up new frontiers for advanced machine learning and computer vision techniques.

5.2 Achievements

In the relentless pursuit of enhancing the capabilities of the robotic system, this research has marked several significant milestones. Foremost among these accomplishments is the precise detection and measurement of joint angles within the robotic arm. This achievement was made possible through the utilization of ArUco markers, distinguished by their distinctive visual patterns that facilitate identification and tracking. ArUco markers emerged as invaluable assets, endowing the system not only with the capacity to recognize the presence of the robotic arm but also with the precision to discern the angles of its individual joints accurately. This breakthrough empowers robots with the ability to comprehensively comprehend their spatial orientation and position within their environment—an advancement with profound implications across a spectrum of applications, from industrial automation to healthcare.

Another notable milestone resides in the domain of object tracking, with a particular emphasis on tracking the robotic arm itself. The research demonstrated exceptional accuracy in tracking the arm's movements, even in the face of challenging environmental factors. Occlusions, varying distances, and complex backgrounds posed formidable challenges, yet the system consistently exhibited its prowess in maintaining accurate tracking. This achievement amplifies the robotic arm's situational awareness and enhances its capacity to interact with objects and navigate intricate, dynamic environments.

5.3 Improvements

While the accomplishments are commendable, the research findings have also unveiled areas primed for further refinement and exploration. Prominent among these areas is the occurrence of false positives in angle detection. To address this challenge effectively, a multifaceted approach can be contemplated.

Improving lighting conditions is crucial to reduce false positives. By optimizing lighting, markers can remain highly visible and easily distinguishable. Thoughtful placement of lighting sources and the use of advanced illumination techniques can significantly enhance marker detection accuracy.

Expanding and diversifying the dataset employed for model training presents an avenue to fortify angle detection. A more comprehensive dataset, spanning diverse scenarios and conditions, can bolster the model's robustness and capacity to discern positive and negative angles with heightened precision.

Exploring different ArUco marker sizes and configurations can be beneficial. Testing various marker sizes and arrangements can provide insights into the optimal configuration for improved detection accuracy.

The exploration of advanced image processing techniques holds the promise of further refining angle detection and curtailing false positives. Techniques such as noise reduction, feature enhancement, and advanced filtering algorithms can be leveraged to elevate the quality of image data, facilitating more accurate angle estimation.

Furthermore, the research should focus on further optimizing the network and data transfer pipelines to reduce the impact of network instability on the overall system performance. This optimization would potentially make the system more robust even in hotspot-based or otherwise constrained network conditions.

Finally, the optimization of camera capabilities is pivotal. The research can delve into the adoption of high-quality cameras characterized by enhanced resolution and heightened sensitivity to markers. Advanced camera models, equipped with state-of-the-art sensors, can augment marker detection, even in demanding conditions such as low-light environments or situations featuring rapidly changing distances.

In conclusion, despite the myriad areas for potential improvements identified throughout this research, it is imperative to underscore that the implemented system fulfils its initial objective. While acknowledging the considerable scope for further refinement and optimization, the system's showcased proficiency in robotic pose estimation and tracking, alongside its commendable accuracy, firmly establish its suitability for practical real-world applications. This research serves as a cornerstone for prospective advancements in the domains of robotics and computer vision, offering the prospect of enhanced precision and a brighter future for human-robot interaction and automation.

References

[1] G. M. H. N. H. Ali, “Kalman Filter Tracking,” *International Journal of Computer Applications,* 2014.

[2] M. P. a. A. Calway, “Real-Time Camera Tracking Using a Particle Filter,” University of Bristol, UK, 2005.

[3] J. G. M. S. M. C. D. P. F. H. J. C. M. Marrón, Comparing a Kalman Filter and a Particle Filter in a Multiple Objects Tracking Application.

[4] L. Tan, Comparison of RetinaNet, SSD, and YOLO v3 for real-time pill identification, 2021.

[5] Á. S. A. B. M. Á. D. S. J. F. V. Ángel Morera, SSD vs. YOLO for Detection of Outdoor Urban Advertising Panels under Multiple Variabilities, 2020.

[6] S. L. ,.-W. S. Y. W.-S. C. a. S.-J. K. Gwon Hwan An, “Charuco Board-Based Omnidirectional Camera Calibration Method,” 2018.

[7] R. S. S. &. M. S. Reilink, “3D position estimation of flexible instruments: marker-less and marker-based methods,” 2012.

[8] S. C. A. A. C. W. N. V. Michail Kalaitzakis, Experimental Comparison of Fiducial Markers for Pose Estimation, 2020.

[9] S. K. a. Z. Bingul, Robot Kinematics: Forward and Inverse Kinematics, 2006.

[10] H. W. L. L. J. .. I. ,. e. a. Hadfield, “The Forward and Inverse Kinematics of a Delta Robot.,” 2020.

[11] A. A. S. Kaushal Dholariya, Conceptual Design and Kinematic Analysis of Humanoid Robot, 2016.

[12] V. S. a. A. T. M. Noman, “Object Detection Techniques: Overview and Performance Comparison,” Ajman, 2019.

[13] J. M. a. S. B. R. Phadnis, “Objects Talk - Object Detection and Pattern Tracking Using TensorFlow,” Coimbatore, 2018.

[14] S. P. X. Y. a. Q. N. John Estrada, Deep-Learning-Incorporated Augmented Reality Application for Engineering Lab Training, 2022.

[15] B. T. Zoltan Siki, Automatic Recognition of ArUco Codes in Land Surveying Tasks, 2021.

[16] P. I. Corke, “A Simple and Systematic Approach to Assigning Denavit–Hartenberg Parameters,” 2007.

[17] A. S. S. S. Amanpreet Singh, D-H Parameters Augmented with Dummy Frames for Serial Manipulators Containing Spatial Links, 2014.

[18] O. G. Anton Larsson, Comparative Analysis of the Inverse Kinematics of a 6-DOF Manipulator, 2023.

Appendix A Marker Generation Script

```
import numpy as np
import cv2
import os
import cv2.aruco as aruco

def generate_aruco_marker(marker_id, markerSize, marker_size_cm, resolu-
tion_ppcm, image_path):
    # Calculate the marker size in pixels at the specified resolution
    marker_size_pixels_res = int(marker_size_cm * resolution_ppcm)

    # Create a dictionary with the desired ArUco marker parameters
    key = getattr(cv2.aruco,f'DICT_{markerSize}X{markerSize}_1000')
    aruco_dict = aruco.getPredefinedDictionary(key)

    # Create an image with the ArUco marker
    marker_image = aruco.drawMarker(aruco_dict, marker_id,
marker_size_pixels_res)

    # Save the marker image to a file
    cv2.imwrite(image_path, marker_image)


def generate_white_box(box_size_cm, resolution_ppcm, output_path):
    # Calculate the box size in pixels
    box_size_pixels = int(box_size_cm * resolution_ppcm)

    # Create a white box image
    box_image = np.ones((box_size_pixels, box_size_pixels),
dtype=np.uint8) * 255

    # Save the white box image to a file
    cv2.imwrite(output_path, box_image)
    print("White box generation complete!")

# White box generation settings
box_size_cm = 5.4
white_res_ppcm = 37.84
output_path = "3_Markers Design/white_box.png"

# Marker generation settings
markerSizeStart = 6
markerSizeEnd = 6
num_markers = 20
marker_size_cm = 5
marker_res_ppcm = 37.8
```

```
# Generate white box
generate_white_box(box_size_cm, white_res_ppcm, output_path)

# Generate markers
os.makedirs("3_Markers Design/Generated_Markers", exist_ok=True)

for i in range(markerSizeEnd+1):
    if i >= markerSizeStart:
        for j in range(num_markers):
            marker_id = j
            marker_path = f"3_Markers Design/Generated_Mark-
ers/{i}x{i}_marker_{j}.png"
            generate_aruco_marker(marker_id, i, marker_size_cm,
marker_res_ppcm, marker_path)
            print(f"Generated {i}x{i}_marker_{j}")

# Generate box markers
os.makedirs("3_Markers Design/Generated_Box_Markers", exist_ok=True)

for i in range(markerSizeEnd+1):
    if i >= markerSizeStart:
        for j in range(num_markers):
            # Load the images
            background = cv2.imread("3_Markers Design/white_box.png")
            marker_path = f"3_Markers Design/Generated_Mark-
ers/{i}x{i}_marker_{j}.png"
            overlay = cv2.imread(marker_path)

            # Extract the dimensions of the image
            height_b, width_b, channels_b = background.shape
            height_o, width_o, channels_o = overlay.shape

            # ROI Algorithm
            y , x = tuple((background.shape[i] - overlay.shape[i]) // 2
for i in range(2))
            h , w = overlay.shape[:2]

            # Create a mask
            mask = np.zeros(background.shape[:2], dtype=np.uint8)
            mask[y:y+h, x:x+w] = 255

            # Loop over each pixel in the background image and update
based on the mask
            for k in range(background.shape[0]):
                for v in range(background.shape[1]):
                    if mask[k, v] != 0:
                        background[k, v] = overlay[k-y, v-x]

            # Save the modified image
```

```
            box_marker_path = f"3_Markers Design/Generated_Box_Mark-
ers/{i}x{i}_marker_{j}.png"
            cv2.imwrite(box_marker_path, background)
            print(f"Generated {i}x{i}_box_marker_{j}")
```

Appendix B – Camera Calibration Script

```
import cv2
import numpy as np

# Define the size of the Charuco board (in squares)
squaresX = 8
squaresY = 6
squareLength = 0.04  # length of each square (in meters)
markerLength = 0.02  # length of each marker (in meters)

# Create the Charuco board
board = cv2.aruco.CharucoBoard_create(squaresX, squaresY, squareLength,
markerLength, cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_250))

# Capture images of the Charuco board
img_paths = ['image1.png', 'image2.png', 'image3.png', ...]  # list of
image paths
objpoints = []  # list of object points
imgpoints = []  # list of image points
for img_path in img_paths:
    # Load the image
    img = cv2.imread(img_path)

    # Convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detect the markers of the Charuco board in the image
    corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(gray,
cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_250))
    charuco_corners, charuco_ids, _ = cv2.aruco.interpolateCorner-
sCharuco(corners, ids, gray, board)

    # Collect the object points and image points
    if len(charuco_corners) > 0:
        objpoints.append(board.objPoints)
        imgpoints.append(charuco_corners)

# Calibrate the camera
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints,
gray.shape[::-1], None, None)

# Save the camera matrix and distortion coefficients to a file
np.savetxt('camera_matrix.txt', mtx)
np.savetxt('distortion_coefficients.txt', dist)
```

Appendix C – Client Webcam Feed Script

```
import asyncio
import cv2
import numpy as np
import time
import websockets

async def receive_frames():
    async with websockets.connect('ws://100.77.189.76:8765') as web-
socket:
        cap = cv2.VideoCapture(0)
        start_time = time.time()
        frames = 0

        while True:
            # Receive the frame from the websocket
            data = await websocket.recv()

            # Convert the bytes to a NumPy array
            buffer = np.frombuffer(data, dtype=np.uint8)

            # Decode the JPEG image
            frame = cv2.imdecode(buffer, cv2.IMREAD_COLOR)

            # Calculate the elapsed time and FPS
            elapsed_time = time.time() - start_time
            fps = frames / elapsed_time

            # Display the FPS counter on the image
            fps_text = f"FPS: {fps:.2f}"
            cv2.putText(frame, fps_text, (10, 30), cv2.FONT_HERSHEY_SIM-
PLEX, 1, (0, 255, 0), 2)

            # Show the image
            cv2.imshow('Frame', frame)

            # Close all windows when q pressed
            if cv2.waitKey(1) & 0xFF == ord('q'):
                break

            # Increment the frame counter
            frames += 1

        cv2.destroyAllWindows()

# Start the client to receive frames
asyncio.get_event_loop().run_until_complete(receive_frames())
```

Appendix D – Server Webcam Feed Script

```
import asyncio
import cv2
import numpy as np
import websockets

async def video_feed(websocket, path):
    cap = cv2.VideoCapture(0)

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Convert the frame to bytes
        _, buffer = cv2.imencode('.jpg', frame)
        data = buffer.tobytes()

        # Send the frame over the websocket
        await websocket.send(data)

        print("working")


    cap.release()

# Start the websocket server
start_server = websockets.serve(video_feed, '100.77.189.76', 8765) #hp
#start_server = websockets.serve(video_feed, '100.89.155.88', 8765)
#epyc

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Appendix E – Dataset Pipeline & Training Script

```
import os
import glob
import xml.etree.ElementTree as ET
import pandas as pd
import zipfile
import shutil
import random

# Step 1: ADD images.zip and annotations.zip to customTF2
# Step 2: Enter Virtual Enviroemnt name
# Step 3: Enter the path for the python vm with tensorflow installed in
anacondaPath
# Step 4: Run File

#user parameters
virtenv = "test"
anacondaPath  = "D:/anaconda3/envs/" + virtenv + "/python.exe"

def extract():
  path_to_zip_file1 = "customTF2/images.zip"
  path_to_zip_file2 = "customTF2/annotations.zip"
  directory_to_extract_to = "customTF2/data"

  with zipfile.ZipFile(path_to_zip_file1, 'r') as zip_ref:
      zip_ref.extractall(directory_to_extract_to)

  with zipfile.ZipFile(path_to_zip_file2, 'r') as zip_ref:
      zip_ref.extractall(directory_to_extract_to)

def split():
    test_path = "customTF2/data/test_labels"
    train_path = "customTF2/data/train_labels"

    if not os.path.exists(test_path):
        os.mkdir(test_path)

    if not os.path.exists(train_path):
        os.mkdir(train_path)

    src_folder = r"customTF2/data/annotations"
    dst_folder1 = r"customTF2/data/test_labels"
    dst_folder2 = r"customTF2/data/train_labels"

    pattern = "\*.xml"
    files = glob.glob(src_folder + pattern)
```

```
    totalDataSet = 0
    for path in os.listdir(src_folder):
        if os.path.isfile(os.path.join(src_folder, path)):
            totalDataSet += 1

    randList =  random.sample(range(totalDataSet), totalDataSet)

    i = 0

    for file in files:

        if( i <= (totalDataSet * 0.2)):
            file_name = os.path.basename(files[randList[i]])
            shutil.move(files[randList[i]], dst_folder1 + "/" +
file_name)
            print('Moved to test:', files[randList[i]])

        elif(i > (totalDataSet * 0.2)):
            file_name = os.path.basename(files[randList[i]])
            shutil.move(files[randList[i]], dst_folder2 + "/" +
file_name)
            print('Moved to train:', files[randList[i]])

        i = i + 1

def xml_to_csv(path): #path -> customTF2/data/
  classes_names = []
  xml_list = []

  for xml_file in glob.glob(path + '/*.xml'):
    tree = ET.parse(xml_file)
    root = tree.getroot()
    for member in root.findall('object'):
      classes_names.append(member[0].text)
      value = (root.find('filename').text  ,
               int(root.find('size')[0].text),
               int(root.find('size')[1].text),
               member[0].text,
               int(member[4][0].text),
               int(member[4][1].text),
               int(member[4][2].text),
               int(member[4][3].text))
      xml_list.append(value)

  column_name = ['filename', 'width', 'height', 'class', 'xmin', 'ymin',
'xmax', 'ymax']
  xml_df = pd.DataFrame(xml_list, columns=column_name)
  classes_names = list(set(classes_names))
  classes_names.sort()
```

```
  return xml_df, classes_names

def main_xml_to_csv():
  for label_path in ['customTF2/data/train_labels', 'cus-
tomTF2/data/test_labels']:
    image_path = os.path.join(os.getcwd(), label_path)
    xml_df, classes = xml_to_csv(label_path)
    xml_df.to_csv(f'{label_path}.csv', index=None)
    print(f'Successfully converted {label_path} xml to csv.')

  label_map_path = os.path.join("customTF2/data/label_map.pbtxt")
  pbtxt_content = ""

  for i, class_name in enumerate(classes):
      pbtxt_content = (
          pbtxt_content
          + "item {{\n    id: {0}\n    name: '{1}'\n}}\n\n".format(i +
1, class_name)
      )
  pbtxt_content = pbtxt_content.strip()
  with open(label_map_path, "w") as f:
      f.write(pbtxt_content)
      print('Successfully created label_map.pbtxt ')

def genTF():    #not working with correct interpreter
    t1 = anacondaPath + " customTF2/tfrec.py customTF2/data/test_la-
bels.csv  customTF2/data/label_map.pbtxt customTF2/data/images/ cus-
tomTF2/data/test.record"
    t2 = anacondaPath + " customTF2/tfrec.py customTF2/data/train_la-
bels.csv  customTF2/data/label_map.pbtxt customTF2/data/images/ cus-
tomTF2/data/train.record"

    os.system(t1)
    os.system(t2)

def train():
    training_dir = "customTF2/training"

    if not os.path.exists(training_dir):
        os.mkdir(training_dir)

    train = anacondaPath + " models/research/object_detec-
tion/model_main_tf2.py --pipeline_config_path=customTF2/data/ssd_mo-
bilenet_v2_fpnlite_320x320_coco17_tpu-8.config --model_dir=cus-
tomTF2/training --alsologtostderr"
    os.system(train)

def test():
```

```
    test = anacondaPath + " models/research/object_detection/ex-
porter_main_v2.py --trained_checkpoint_dir=customTF2/training --pipe-
line_config_path=customTF2/data/ssd_mobilenet_v2_fpn-
lite_320x320_coco17_tpu-8.config --output_directory customTF2/data/in-
ference_graph"
    os.system(test)

def gen_files():
    extract()
    split()
    main_xml_to_csv()
    genTF()

def model_and_export():
    train()
    test()
    print("\n\nDone.")

# gen_files()
model_and_export()
```

Appendix F – Tracking Script

```
import time
from turtle import distance
import tensorflow as tf
import cv2
import numpy as np
#from PIL import Image
from object_detection.utils import label_map_util
#from object_detection.utils import visualization_utils as viz_utils
#from base64 import b64encode

PATH_TO_SAVED_MODEL = "customTF2/data/inference_graph/saved_model"

Known_distance = 223
Known_width    = 35 #109 pixels

# Load label map and obtain class names and ids
category_index=label_map_util.create_category_index_from_labelmap("cus-
tomTF2/data/label_map.pbtxt",use_display_name=True)

def visualise_on_image(image, bboxes, labels, scores, thresh):
    (h, w, d) = image.shape
    for bbox, label, score in zip(bboxes, labels, scores):
        if score > thresh:
            xmin, ymin = int(bbox[1]*w), int(bbox[0]*h)
            xmax, ymax = int(bbox[3]*w), int(bbox[2]*h)

            cv2.rectangle(image, (xmin, ymin), (xmax, ymax), (0,255,0),
2)
            cv2.putText(image, f"{label}: {int(score*100)} %", (xmin,
ymin), cv2.FONT_HERSHEY_SIMPLEX, 1, (255,255,255), 2)

            #cv2.putText(frame, "xmin: " + str(xmin) + " xmax: " +
str(xmax) + " ymin: " + str(ymin) + " ymax: " + str(ymax), (50,50),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,0,0), 1)
    return image

def coord(image, bboxes, labels, scores, thresh):
    (h, w, d) = image.shape
    for bbox, label, score in zip(bboxes, labels, scores):
        if score > thresh:
            xmin, ymin = int(bbox[1]*w), int(bbox[0]*h)
            xmax, ymax = int(bbox[3]*w), int(bbox[2]*h)
            return xmin, xmax, ymin, ymax
        else: return 0,0,0,0

def width(image, bboxes, labels, scores, thresh):
```

```
    (h, w, d) = image.shape
    for bbox, label, score in zip(bboxes, labels, scores):
        if score > thresh:
            xmin, ymin = int(bbox[1]*w), int(bbox[0]*h)
            xmax, ymax = int(bbox[3]*w),
int(bbox[2]*h)
            return (xmax - xmin)
        else: return 0 #1 to not cause non div error

# focal length finder function

def FocalLength(measured_distance, real_width, width_in_rf_image):
    focal_length = (width_in_rf_image* measured_distance)/ real_width
    return focal_length

#camera at height of 80-90 84cm
def Distance_finder(Focal_Length, real_face_width, face_width_in_frame):
    distance = (real_face_width * Focal_Length)/face_width_in_frame
    return distance


if __name__ == '__main__':

    # Load the model
    print("Loading saved model ...")
    detect_fn = tf.saved_model.load(PATH_TO_SAVED_MODEL)
    print("Model Loaded!")

    # Video Capture (video_file)
    #video_capture = cv2.VideoCapture("input.mp4")
    video_capture = cv2.VideoCapture(0)
    start_time = time.time()

    frame_width = int(video_capture.get(3))
    frame_height = int(video_capture.get(4))
    #fps = int(video_capture.get(5))
    size = (frame_width, frame_height)

    #Initialize video writer
    result = cv2.VideoWriter('result.avi', cv2.Vide-
oWriter_fourcc(*'MJPG'),15, size)

    #-----Ref Data Gathering-------
    ref_image = cv2.imread("ref_images/ref1.jpg")
    ref_image_face_width = 109#width(ref_image)
    focal_length_found = FocalLength(Known_distance, Known_width,
ref_image_face_width)
    print("Focal Length of Ref: ",focal_length_found)
    #cv2.imshow("ref_image", ref_image)
```

```
    while True:
        ret, frame = video_capture.read()
        if not ret:
            print('Unable to read video / Video ended')
            break

        frame = cv2.flip(frame, 1)
        image_np = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        # The input needs to be a tensor, convert it using `tf.con-
vert_to_tensor`.
        # The model expects a batch of images, so also add an axis with
`tf.newaxis`.
        input_tensor = tf.convert_to_tensor(image_np)[tf.newaxis, ...]

        # Pass frame through detector
        detections = detect_fn(input_tensor)

        # Set detection parameters
        score_thresh = 0.4   # Minimum threshold for object detection
        max_detections = 1

        # All outputs are batches tensors.
        # Convert to numpy arrays, and take index [0] to remove the
batch dimension.
        # We're only interested in the first num_detections.
        scores = detections['detection_scores'][0, :max_detec-
tions].numpy()
        bboxes = detections['detection_boxes'][0, :max_detec-
tions].numpy()
        labels = detections['detection_classes'][0, :max_detec-
tions].numpy().astype(np.int64)
        labels = [category_index[n]['name'] for n in labels]

        # Display detections
        visualise_on_image(frame, bboxes, labels, scores, score_thresh)
        #-----------------------------------------------------------------
----------------------

        #if not coord(frame, bboxes, labels, scores, score_thresh) ==
(0,0,0,0):
            #print(coord(frame, bboxes, labels, scores, score_thresh))

        # if not width(frame, bboxes, labels, scores, score_thresh) ==
0:
        #     print("Width: ", width(frame, bboxes, labels, scores,
score_thresh))

        # measured_distance = 223
```

```
        # real_width = 35         #width of robot wheel to wheel
        # real_face_width = 35    #width of robot wheel to wheel #may
need to replace this for another value

        # face_width_in_frame = width(frame, bboxes, labels, scores,
score_thresh)
        # width_in_rf_image    = 136#width(frame, bboxes, labels, scores,
score_thresh) #need to fix this

        # Focal_Length = FocalLength(measured_distance, real_width,
width_in_rf_image)
        # Distance = Distance_finder(Focal_Length, real_face_width,
face_width_in_frame)

        # print("Focal Length: ", Focal_Length)
        # print("Distance: ", Distance)

        # calling face_data function
        face_width_in_frame = width(frame, bboxes, labels, scores,
score_thresh)
        # finding the distance by calling function Distance
        if face_width_in_frame != 0:
            Distance = Distance_finder(focal_length_found, Known_width,
face_width_in_frame)
            # Drwaing Text on the screen
            cv2.putText(frame, f"Distance: {Distance} cm", (frame_width
- 270, frame_height - 20), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,0),
2)

        #-------------------------------------------------------------
----------------------
        end_time = time.time()
        fps = int(1/(end_time - start_time))
        start_time = end_time

        #cv2.imwrite("images/ref1.jpg",frame)

        cv2.putText(frame, f"FPS: {fps}", (20,frame_height-20),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,0), 2)
        #cv2.putText(frame, f"BoxWidth: {face_width_in_frame}",
(frame_width - 250, frame_height - 120), cv2.FONT_HERSHEY_SIMPLEX, 1,
(0,0,0), 2)
        #cv2.putText(frame, f"Focal  L: {Focal_Length}", (frame_width -
250, frame_height - 60), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,0,0), 2)


        #cv2.imwrite("images/ref2.jpg",frame)

        #Write output video
```

```
        result.write(frame)

        cv2.imshow("Results", frame)

        key = cv2.waitKey(1) & 0XFF
        if key == ord("q"):
            break

video_capture.release()
```

Appendix G – Marker Pose Detection Script

```
import numpy as np
import cv2
import time

# Load the dictionary and parameters
dictionary = cv2.aruco.getPredefinedDictionary(cv2.aruco.DICT_6X6_100)
parameters = cv2.aruco.DetectorParameters_create()

# Define the size of the markers (in meters)
marker_size = 0.05

# Define the camera matrix and distortion coefficients
camera_matrix = np.array([[6.73172250e+02, 0.00000000e+00,
3.21652381e+02],
                          [0.00000000e+00, 6.73172250e+02,
2.40854103e+02],
                          [0.00000000e+00, 0.00000000e+00,
1.00000000e+00]])
distortion_coefficients = np.array([-2.87888863e-01, 9.67075352e-02,
1.65928771e-03, -5.19671229e-04, -1.30327183e-02])

# Initialize the camera
cap = cv2.VideoCapture(0)

while True:
    # Capture a frame from the camera
    ret, frame = cap.read()

    # Convert the frame to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # Detect the markers in the frame
    corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(gray, dic-
tionary, parameters=parameters)

    # If markers are detected, estimate their pose
    if ids is not None:
        rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(corners,
marker_size, camera_matrix, distortion_coefficients)

        # Draw the axes of the markers
        for i in range(len(ids)):
            cv2.aruco.drawDetectedMarkers(frame, corners, ids)
            cv2.aruco.drawAxis(frame, camera_matrix, distortion_coeffi-
cients, rvecs[i], tvecs[i], marker_size)
```

```
    # Check for the 'q' key to exit
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

# Release the camera and close the window
cap.release()
cv2.destroyAllWindows()
```