Semester project on signals in robot systems

BEng in Robot Systems

3. Semester

Authors (Group 3)

Name	Username	Date of birth
Noah V. Vryens	novry24	01/04/2002
Rasmus K. Nielsen	rasni19	25/12/1997
Emma B. Rasmussen	erasm24	15/05/2001
Jannick H. Irvold	jairv24	23/10/2002
Eymundur Ó. Pálsson	eypal24	31/08/2002

Supervisor

Thorbjørn M. Iversen

thmi@mmmi.sdu.dk

December 16, 2025

Abstract Placeholder

This report documents the design, implementation, and evaluation of an autonomous robotic system capable of identifying a target and throwing a projectile to hit it using a Universal Robots UR5 manipulator. The system addresses the challenges of dynamic object manipulation by integrating robust machine vision, ballistic trajectory planning, and real-time kinematic control.

The vision subsystem, developed in C++ using the OpenCV and Pylon libraries, employs a Basler camera to capture the workspace. It utilizes intrinsic camera calibration and homography transformation matrices to accurately map 2D pixel coordinates to the robot's 3D Cartesian frame. Target detection is achieved through image rectification and the Hough Circle Transform, allowing for the precise localization of circular targets on a planar surface.

For trajectory generation, a physics-based ballistic model calculates the required release velocity and launch angles (yaw and pitch) to reach the target coordinates. These parameters are processed by a MATLAB computational backend, which leverages the Robotics System Toolbox to model the UR5 kinematics. The system solves the inverse kinematics for the release configuration using a Generalized Inverse Kinematics (GIK) solver with strict joint and position constraints. To ensure the end-effector achieves the exact velocity vector required for the throw, a weighted Jacobian-based velocity controller generates the motion profile, incorporating a computed lead-up phase to account for joint acceleration limits and a follow-through phase to maintain orientation.

Robot control is executed via the ur_rtde interface, which establishes a real-time control loop (125 Hz) with the UR5 controller. The system utilizes speedJ commands to execute the generated joint-space velocities, ensuring smooth and dynamic motion. Synchronization with a custom-actuated gripper is handled through a TCP/IP socket connection, triggering the release at the precise calculated moment. This report details the software architecture, mathematical modelling of the kinematics and dynamics, and the experimental results regarding the system's throwing accuracy and repeatability.

Contents

1 Introduction

As the robotics industry develops, it is becoming increasingly common to see robots used to solve both important, difficult and dangerous tasks. Dealing with real world problems is often the goal when inventors work towards new technologies. Some inventions may be used in a multitude of ways and further the development of other increasingly complex solutions, while others serve society in small but significant ways for decades to come. As the technologies mature and become more widely available, new possibilities arise. Opportunities to use the technology for less serious tasks and instead focus on showing its capabilities in a fun and eye-catching way. This could be done by using it in an unconventional environment or by completing a task far outside the intent of the inventor. This project will try to use technology in such a way. It will try to use methods originally developed for the industry to complete an objective it was not intended for.

1.1 Project Goals

Using a UR5 robot arm with an attached gripper, a camera, and software, the aim of this project is to pick up a ball and throw it at a dart board placed within view of the camera. For safety reasons the ball thrown is light weight, made of plastic and is about the size of a ping pong ball. To allow it to stick to the dart board, it is coated with Velcro, just as the dart board is. Success will be measured based on how likely a throw is to be within the bullseye.

1.2 Problem Definition

Making a UR5 robot arm throw an object at a designated target

- Identify the target point using machine vision
- Plan a trajectory for the object using physics and kinematics
- Control a UR5 and gripper based on the modeled trajectory

The solution is expected to consistently hit within the visual bullseye of the Velcro dart board, as seen in figure 1.1. Specifically, the center of the ball should be within 2 cm of the center of the bullseye. This accuracy would also allow the solution to hit within the opening of a standard American 16 oz red solo cup with a ping pong ball. The accuracy is accepted to be consistent, if at least 90% of throws hit within the accepted target area. In the spirit of the project, a throw will be defined such that it must have an upwards initial trajectory and must cover at least 20cm in horizontal distance.

1.3 Constraints & Limitations

The project is subject to constraints originating from physical limitations, and limitations set both externally and internally. Safety configurations limit the robot's motion in both joint and Cartesian space, while the project goal is limited to throwing in one general direction rather than 360 degrees around the robot.

As only one camera is available at each robot cell, depth perception is limited and thus all targets are assumed to be located at table height. All software will be developed and tested in a Linux based environment, support for any other operating system will not be considered.

Figure 1.1: The Velcro Dart Board

1.4 System Overview

The system developed in this project integrates several components that enable the robot to perceive its environment, plan feasible trajectories, and execute accurate motions within a calibrated workspace. The system is structured around five core modules: machine vision, calibration, trajectory planning, kinematics, and robot control, which operate sequentially and are closely interconnected.

The machine vision module captures images of the table surface and detects the center of the target on the dartboard. This position is initially expressed in the table frame and subsequently transformed into world-frame coordinates. The resulting target position is then used as input for the subsequent planning and control stages.

Trajectory planning generates a feasible motion from the robot's current configuration to the target location while respecting kinematic and workspace constraints. Kinematic computations are used to convert the planned end-effector motion into corresponding joint configurations through inverse kinematics, with forward kinematics used for verification.

The robot control module executes the planned trajectory by commanding the robot joints to follow the desired motion profile. Together, these modules form a modular and integrated system that enables reliable operation within the table workspace. An overview of the complete experimental setup is shown in Figure 1.2.

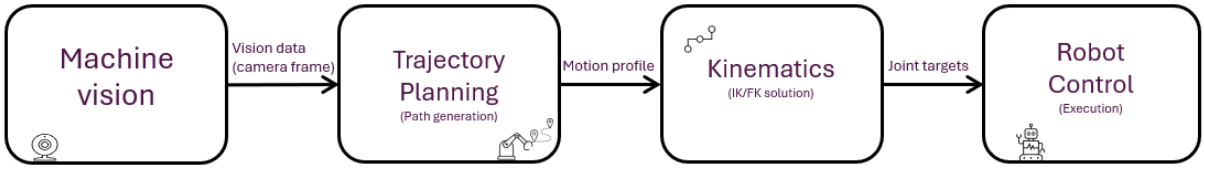


Figure 1.2: System overview of the robotic throwing setup.

2 Machine Vision

Since the robot needs to be able to hit a specific target which can be in different places, a form of machine vision is needed. In particular some sort of algorhithm which can find the defined target. A Basler Pylon (Model Nr. acA 1440-220uc with C125-0418-5M lens) camera is used for this purpose, this makes it possible to easily integrate it into the C++ code over a USB conection using Basler Pylons libraries, together with OpenCV.

2.1 Calibration

The camera is modeled using the pinhole-model. This takes every point in 3D space and projects it through a single point, the pinhole, and hits the image plane forming a 2D image. However, in the real world light bends going through the lense so this needs to be modeled as well.

This is done using the cameras intrinsic matrix which contains the vertical and horizontal focal lengths and the cameras centerpoint (pinhole), together with five distortion coefficients to account for radial and tangential distortion.

OpenCV models radial distortion using the following formula:

$$x_{corrected} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.1}$$

$$y_{corrected} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.2}$$

Radial distortion is caused by the curve of the lense and shows up as curving of the image the further from the center one looks.

OpenCV models tangential distortion using the following formula:

$$x_{corrected} = x + [2p_1 xy + p_2(r^2 + 2x^2)] \tag{2.3}$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2 xy] \tag{2.4}$$

Tangential distortion is caused by the lense being on an angle to the image plane, making the image look skewed in one direction.

OpenCV then combines these into a 1 by 5 matrix:

$$Distortion_{coefficients} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix} \tag{2.5}$$

Together with the intrinsic matrix:

$$CameraMatrix = \begin{pmatrix} f_x & 0 & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{2.6}$$

Where f_x, f_y is the focal lengths and x_0, y_0 is the pinhole offset.

To calculate all of these parameters, a standard already known method is used [1]. This uses multiple images of a chessboard to find points with known distances between them in the cameras field of view to calculate the intrinsic camera matrix and lens distortion coefficients. However, these parameters do not correct the perspective shift from the camera being at an angle to the table, the homography must be computed as well. Though, as only the table plane is relevant this becomes somewhat relatively simple.

2.2 Homography

Since only the table plane is relevant, a linear mapping from the image coordinates to the table coordinates can be computed. This mapping is called a homography, **H**.

The relationship is best described using homogeneous coordinates, where a 2D point is represented with an added third dimension equal to 1, also known as the scale factor. The homography relates a point in the image plane $\mathbf{x}'$ to a point on the table plane $\mathbf{x}$ as: $\alpha\mathbf{x} = \mathbf{H}\mathbf{x}'$

Writing this out with coordinates (i', j') for the image and (i, j) for the table, the following system of equations can be written up:

$$\begin{pmatrix} i \cdot w \\ j \cdot w \\ w \end{pmatrix} = \begin{pmatrix} p_{00} & p_{01} & p_{02} \\ p_{10} & p_{11} & p_{12} \\ p_{20} & p_{21} & 1 \end{pmatrix} \begin{pmatrix} i' \\ j' \\ 1 \end{pmatrix} \tag{2.7}$$

Since the scaling factor can be chosen freely, it is possible to set the last entry of the matrix p_{22} to 1. This leaves 8 unknown entries in the matrix that needs to be found.

To solve for these unknowns, the scale factor w can be eliminated by substituting the third equation ($w = p_{20}i' + p_{21}j' + 1$) into the first two. This results in the following two equations for a single point:

$$i = p_{00} \cdot i' + p_{01} \cdot j' + p_{02} - p_{20} \cdot i \cdot i' - p_{21} \cdot i \cdot j' \tag{2.8}$$

$$j = p_{10} \cdot i' + p_{11} \cdot j' + p_{12} - p_{20} \cdot j \cdot i' - p_{21} \cdot j \cdot j' \tag{2.9}$$

Since each pair of corresponding points $(i', j') \leftrightarrow (i, j)$ gives two equations, and there are 8 unknowns, at least four point pairs are needed to solve the system. This is set up as a matrix equation $\mathbf{A}\mathbf{p} = \mathbf{b}$ [1], where $\mathbf{p}$ contains the unknown homography entries. Solving this gets the specific values for the homography matrix **H**, which makes it possible to convert any pixel coordinate in the image to a physical coordinate on the table.

2.3 Finding Circular Objects

Objects in the image are detected using the Hough transform, which is a feature extraction technique commonly used in machine vision. The purpose of this method is to detect geometric shapes that can be described by mathematical equations, such as straight lines or circles, even when the image is noisy or if edges are incomplete. For the purposes of this project, the circle variant is used, as it best describes the defined target.

A circle with a fixed radius r is defined by the equation:

$$(x-a)^2 + (y-b)^2 = r^2 \tag{2.10}$$

Where (x,y) are the coordinates in the image space, and (a,b) is the center of the circle.

When the radius of the circle is known, the Hough transform uses a 2D parameter space corresponding to the possible center coordinates (a,b). An accumulator array of this parameter space is initialized with all values set to zero.

First, an edge detection algorithm is applied to the image to identify edge pixels. For each detected edge pixel (x,y), the algorithm computes all possible center points (a,b) for which a circle with the given radius r passes through (x,y). Each valid center location receives a vote by incrementing the corresponding cell in the accumulator array.

In other words, each edge pixel contributes votes along a circular path in the parameter space. When multiple edge pixels belong to the same physical circle in the image, their votes intersect at a common location in the accumulator. This intersection produces a local maximum, which corresponds to the estimated center of the detected circle. Though the implementation in OpenCV uses a slightly more complex version where the radius r can vary. However, all this does, is expand the parameter space to 3D where the new axis is the different values r can have.

2.4 Implementation with OpenCV

The machine vision system is implemented as a dedicated C++ class, `Vision`, which handles camera access, calibration, and object detection.

2.4.1 Calibration Implementation

The calibration process consists of two stages: intrinsic camera calibration and table plane rectification.

Using `Vision::calibrateCam` to find the intrinsics and distortion coefficients, 20 images are captured, and the inner chessboard corners are detected using `cv::findChessboardCorners`. These corner locations are refined to sub-pixel accuracy using `cv::cornerSubPix` before being passed to `cv::calibrateCamera`. The resulting camera matrix and distortion coefficients are saved to a .yaml file with the tableID in the filename.

Then, to map image coordinates to the physical table plane, the `Vision::calibrateTableCorners` function computes a homography matrix $\mathbf{H}$. The user selects the four corners of the table in the image using a mouse callback. These pixel coordinates are paired with known real-world table coordinates defined in millimeters.

Afterwards, the homography matrix is computed using `cv::findHomography`, now applying `cv::warpPerspect` produces a rectified, top-down view of the table where distances in the image directly correspond to physical

distances on the table. This homography matrix is then also saved to a .yaml file with a tableID.

2.4.2 Object Detection

The circular target object is detected in the `Vision::findCircularObject` method using the following steps:

First, the input image is undistorted using the intrinsic camera matrix together with the distortion coefficients and then rectified using the homography matrix. This results in a top-down view of the table within the four user-defined corners of the table with perspective and distortion removed (this is done before passing it to the function).

Then, the rectified image is converted to grayscale and blurred using a Gaussian filter. This reduces image noise and improves the reliability of edge detection.

At last, circular features are detected using `cv::HoughCircles`. The threshold value for edge detection and the circle confidence value are provided by the main program (typically 50 and 30 respectively). The minimum and maximum radius are set to 150 and 180 pixels to ensure that only the defined target is detected.

2.4.3 Table to World Transformation

The `Vision::tableToWorld` function transforms these coordinates into the world frame. This consists of a rotation matrix **R** and a unit conversion from millimeters to meters:

$$\mathbf{P}_{world} = \mathbf{R} \cdot \begin{pmatrix} c_x \\ c_y \\ 0 \end{pmatrix} \cdot \frac{1}{1000} \tag{2.11}$$

This is done so that the target coordinates are more easily used by MATLAB and the robot.

3 Trajectory Planning

3.1 Physics

The first part of finding the trajectory of an object is to find its path given two coordinates, the release and target coordinate. With the given coordinates yaw, pitch and start velocity ($\mathbf{v}_0$) can be calculated for the release position. Assuming that the release coordinate is in (0. 0. 0) yaw is calculated with $\arctan_2(x,y)$. To calculate $\mathbf{v}_0$ the following function, where θ_z is pitch, is used:

$$v_0 = \sqrt{\frac{g \cdot \sqrt{x+y}}{2\cos^2(\theta_z)(\sqrt{x+y}\tan(\theta_z) - z)}} \tag{3.1}$$

To find the pitch which has the lowest $\mathbf{v}_0$, 1000 pitches spaced linearly between 22.5 and 90 degrees are checked.

3.2 Kinematics

Given the object trajectory required to hit the target, the movement of the robot can be determined such that the object reaches the desired velocity and direction at the point of release. By moving the tool center point (TCP) and there-by the object, the goal is to have the TCP at the exact location of the release point at the same time as the velocity of the TCP matches the desired velocity and direction of the ball trajectory.

To create the movement sequence the desired joint pose and velocity will be determined for a set of time steps spaced 8ms apart to match the robots frequency (f) of 125Hz. The first step in this process is to ensure that the positions and angles given in world frame can be converted to the robots base frame.

3.2.1 Calibration

To enable accurate positioning of the robot relative to points defined on the table, a calibration between the robot base frame and the world frame is required. Since the robot operates in its own coordinate system, a rigid-body transformation must relate world-frame coordinates to robot-frame coordinates, as outlined in standard robot–table calibration methods [2]. This transformation consists of a rotation aligning the coordinate axes and a translation accounting for the offset between the frame origins.

The transformation was estimated using 52 reference points on the table, for which corresponding positions were recorded in both the world frame and the robot frame. A dedicated calibration tool was mounted on the robot tool flange to improve TCP alignment, and the resulting datasets were imported into MATLAB. Prior to estimation, both point sets were centered by subtracting their respective centroids to improve numerical robustness.

The rigid-body transformation was computed using an singular value decomposition (SVD) based least-squares method. The rotation matrix was obtained from the SVD of a correlation matrix constructed from the centered datasets, while the translation vector was computed from the difference between the centroids. The calibration quality was evaluated using the root mean square (RMS) error. The initial calibration resulted in RMS errors between 0.1991 m and 1.3078 m, revealing the presence of inconsistent measurements. After

removing four identified outliers, the calibration was repeated, reducing the RMS error to the range 0.0005 m to 0.0012 m. The results of the initial and repeated calibration procedures are shown in Figures 3.1 and 3.2, respectively.

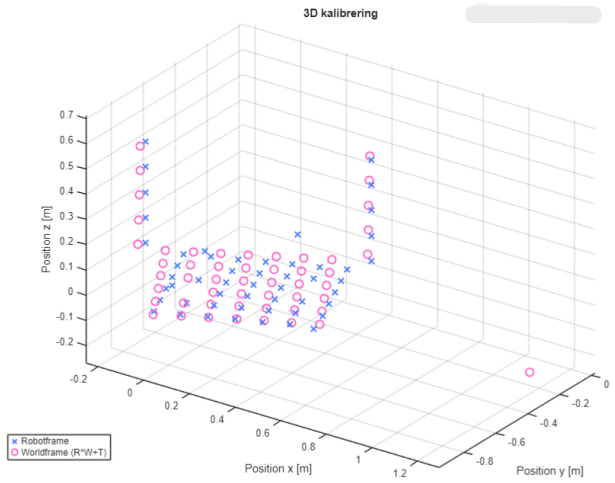


Figure 3.1: Visualization of the initial robot-table calibration using the complete dataset.

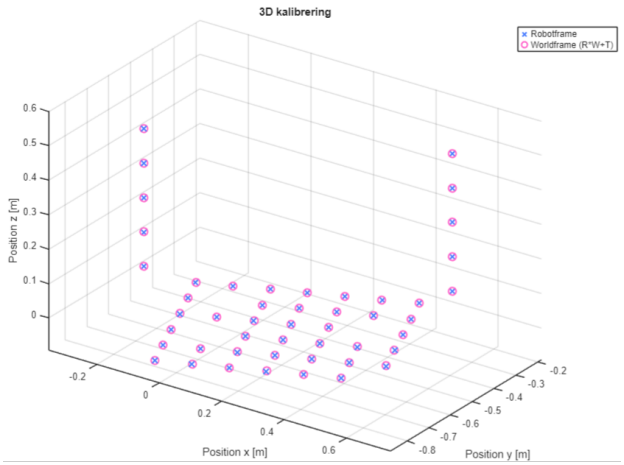


Figure 3.2: Visualization of the repeated robot-table calibration after removal of identified outliers.

3.2.2 Robot Path Planning

With the homogeneous transformation matrix from world frame to base frame found, the robots required path can be determined. This process is done using MATLAB with the Robotics System Toolbox (RST) [4]. The path is split into two sections, what happens before the release point and what happens after. These two path sequences will be called lead-up and follow-through.

Inverse Kinematics

To determine the joint positions for the release point, a generalized inverse kinematics solver from RST is used. This optimization-based solver balances four weighted inputs to find a solution ($\mathbf{q}_{\text{release}}$) within physical limits:

Joint Limits (Weight: 500): Sourced from the UR5 teach pendant, these include a safety offset to prevent the robot from reaching boundaries, which would restrict lead-up and follow-through movement.

Release Position (Weight: 50): The target Cartesian coordinates are transformed from the world frame to the robot's base frame using the calibration matrix.

TCP Orientation (Weight: 1): Derived from the ball's yaw and pitch, the orientation ensures the gripper does not obstruct the ball post-release. A rotation matrix is constructed in the world frame where the Y-axis opposes the trajectory vector, the Z-axis points downward (remaining orthogonal), and the X-axis completes the right-handed frame. This is then transformed to the base frame.

Desired Joint Pose: Set to the center of each joint's movement range to maximize available range of motion in both directions.

The weights are arbitrarily set based on the inputs' relative importance for path planning, with joint limits being the single most important factor. TCP position outweighs orientation, as it is more important to be at the correct position. If a solution is found, it is validated against a 5 mm tolerance; the Euclidean distance between the desired and computed TCP positions. If the solver fails, an error code is generated.

Joint velocity at release

With the joint pose for the release position found, the required joint velocities at release can be determined based on the ball's desired velocity magnitude (*vel*) and direction vector (**dir**). The desired TCP linear velocity is defined as $\mathbf{v}_{\text{release}} = vel \cdot \mathbf{dir}$. First, the Jacobian ($\mathbf{J}$) is found for the joint pose. This 6×6 matrix is derived from the robot's kinematic model using the provided MATLAB function [5]. The first three rows determine the TCP linear velocity, and the last three rows determine the TCP angular velocity. For this task, only the linear velocity portion is necessary, making $\mathbf{J}_{\text{vel}}$ a 3×6 matrix. Since the UR5 is kinematically redundant for this task (6 joints, 3 required outputs), and because certain joints (like Wrist 2) are highly restricted, a weighted pseudo-inverse of the Jacobian is used. This requires defining a diagonal joint cost matrix, $\mathbf{W}$, where higher values impose a greater penalty on the corresponding joint velocity. With the cost matrix, the weighted pseudo-inverse ($\mathbf{J}^{+}_{\text{weighted}}$) is calculated using the following formula:

$$\mathbf{J}^{+}_{\text{weighted}} = \mathbf{W}^{-1}\mathbf{J}^{T}_{\text{vel}}(\mathbf{J}_{\text{vel}}\mathbf{W}^{-1}\mathbf{J}^{T}_{\text{vel}})^{-1} \tag{3.2}$$

With the weighted pseudo-inverse, the required joint velocities at release ($\dot{\mathbf{q}}_{\text{release}}$) can be determined:

$$\dot{\mathbf{q}}_{\text{release}} = \mathbf{J}^{+}_{\text{weighted}}\mathbf{v}_{\text{release}} \tag{3.3}$$

Lead-up

The lead-up path is determined in joint space based on the computed release position ($\mathbf{q}_{\text{release}}$) and the required release velocity ($\dot{\mathbf{q}}_{\text{release}}$). In order to limit the joints' movements, the path is generated linearly for each joint using a constant acceleration (a). The path is computed backwards from the release point until each joint reaches a velocity of 0. For each joint j the number of time steps required to hit the desired velocity is determined:

$$\text{time}_{\mathbf{j}} = \frac{|\dot{\mathbf{q}}_{\text{release},j}|}{a} \tag{3.4}$$

$$\text{steps}_{\mathbf{j}} = \lceil \text{time}_{\mathbf{j}} \cdot f \rceil \tag{3.5}$$

The joint requiring the most steps determines the total steps for the trajectory ($\text{steps}_{\text{max}} = \max(\text{steps}_j)$). Then the position ($\mathbf{q}$) and velocity ($\dot{\mathbf{q}}$) at each time step i can be determined for each joint j using the time step $\Delta t = 1/f$. The iteration index i runs backward from $i = 1$ (the release point) to $i = \text{steps}_{\text{max}}$ (the start of the movement):

$$\dot{\mathbf{q}}_{i,j} = \dot{\mathbf{q}}_{\text{release},j} \cdot \max\left(\left(1 - \frac{i}{\text{steps}_{\mathbf{j}}}\right), 0\right) \tag{3.6}$$

$$\mathbf{q}_{i,j} = \mathbf{q}_{i-1,j} - \dot{\mathbf{q}}_{i,j} \cdot \Delta t \tag{3.7}$$

The $\max(\cdot, 0)$ function ensures that joints which complete their movement faster than the global $\text{steps}_{\text{max}}$ will have their velocity clipped to zero for the remainder of the trajectory, thereby synchronizing the start and end of all joint movements. With the full lead-up sequence determined it can now be checked against the joint limits. Each joint's positions is checked against their respective limits, if any joint exceeds the boundaries, an error code is registered (see section 3.2.2).

Follow-through

The follow-through is determined in Cartesian space rather than joint space to allow the TCP to decelerate in a linear motion in the direction of the ball's initial trajectory. This is done to allow for small errors in the timing of the release. Should the ball be released slightly too late, it will still be traveling in the correct direction at a reduced speed, negating some of the impact of the delayed release. The method for determining the follow-through path is similar to the method used for the lead-up, but since it is done in Cartesian space, some changes are required.

Rather than using a constant deceleration in joint space to compute the require time, the opposite is done here. A static time of 0.5 seconds is set for the follow-through, which generates a static number of time steps ($\text{steps}_{\text{follow}}$). The desired velocity in Cartesian space can then be calculated for each time step such that the TCP decelerates at a constant rate until the velocity is 0 at the last time step in the sequence. At each time step, the desired Cartesian velocity can be converted to joint velocities using that positions Jacobian. The Jacobian is again found using the kinematics function and transformed into a weighted pseudo-inverse ($\mathbf{J}^{+}_{\text{weighted,i}}$) with only the velocity parameters. With the joint velocity, the joint position can be updated. The iteration index i runs from $i = 1$ to $\text{steps}_{\text{follow}}$

$$\text{steps}_{\text{follow}} = round(0.5 \cdot f) \tag{3.8}$$

$$\mathbf{v}_{\text{i}} = (1 - \frac{i}{\text{steps}_{\text{follow}}}) \cdot \mathbf{v}_{\text{release}} \tag{3.9}$$

$$\dot{\mathbf{q}}_{\text{i}} = \mathbf{J}^{+}_{\text{weighted,i}} * \mathbf{v}_{\text{i}} \tag{3.10}$$

$$\mathbf{q}_{\text{i}} = \mathbf{q}_{\text{i-1}} + \dot{\mathbf{q}}_{\text{i}} \cdot \Delta t \tag{3.11}$$

Since the follow-through trajectory is generated in Cartesian space, it is not limiting the amount of space needed in joint space. Therefore, the joints' positions at each time step is checked against their respective limits, and their velocity set to 0 if the joint gets within 5 degrees of the limit. This does mean that some follow-through trajectories will not be linear in Cartesian space for the full movement, but will stay linear for as long as the joints allow.

Safety check

With the lead-up, release point, and follow-through generated the entire sequence can be combined to create the full movement. Since only the joint limits have been checked during the generation of the movement, the TCP limits in Cartesian space must be checked by the end to ensure that movement will not push the TCP outside its boundaries. This is done by finding the TCP position at each time step using forward kinematics and checking its x, y and z coordinate against the limits. If any position is outside the limits, an error code is registered (see section 3.2.2).

Status codes

Path generation utilizes two code types to track safety and accuracy. Error codes (2x) indicate a safety limit breach (e.g., TCP or joint limits) that terminates generation. Status codes (1xxxxx) allow generation to continue but flag imperfections; while 100000 represents a nominal path, each subsequent digit tracks a specific deviation, such as the IK release offset (mm) or velocity loss (%).

- 100000: All good

- 1x0000: IK release point is off by x in mm
- 10x000: Release velocity loss by x in %
- 100100: Joint angles clipped to limits in follow-through
- 20: No IK solution for release configuration
- 21: IK release point is off by > 5mm
- 22: Release velocity loss of > 3%
- 23: Joint limits exceeded in lead-up
- 24: TCP limits exceeded
- 26: Release point too close to target point

Release Points

As the generation of both the ball's trajectory and the robot's movement is determined based on the release point, some release points may lead to a solution while others may not. In order to achieve the highest likelihood that a solution is found, the calculations are run multiple times using a sequence of 7 release points. If the first release points does not generate a valid solution, the next is tried and so on. The 7 points were chosen during the test phase of the system and should be enough to cover most of the target area that is physically possible to hit. If no solution is found for any of the 7 points, it is likely that no release point exists that would generate a solution for that specific target point. MATLAB will return a full status code with all 7 error codes to the C++ program, letting the user know that the target is not within reach, while giving some indication as to what makes the target impossible to hit.

3.2.3 MATLAB Communication

To run the MATLAB script, the MATLAB Engine API is used. This allows the C++ program to start the MATLAB script, pass data over, and return any data from the calculations. MATLAB Engine works by starting the script as a separate process, passing the data through the API thus copying the data in a format MATLAB can understand, running the script through, returning it through the API and lastly converting the data to something C++ can understand and closing the MATLAB process. Because MATLAB Engine relies on its own MATLAB International Components for Unicode (ICU), the program must be started through a shell script, running on top of the default Linux ICU. The reason the MATLAB ICU cannot be loaded solely is because Pylon, the library for vision, requires access to the standard Linux ICU.

4 Robot Control

4.1 Gripper

The gripper used in the project is a WSG50 from Weiss. The connection to the gripper happens with a Transmission Control Protocol (TCP) server and client. The gripper is the TCP server, and the TCP client is made in a C++ class. To control the gripper 4 functions are used. `HOME()`, `GRIP()`, `RELEASE()`, `BYE()`. [3]

- `HOME()` opens the gripper fully when the connection is established, ensuring it always starts in the same position.
- `GRIP()` closes the gripper until a 80N force is reached.
- `RELEASE()` opens the gripper 5mm.
- `BYE()` disconnects the gripper safely.

Each function returns "ACK function name()" when it starts, and returns "FIN function name()" when it is done. Except BYE() which only return "FIN BYE()".

Figure 4.1: WSG50 Weiss gripper

4.2 UR5 Communication Interface

In order to control the robot and simultaneously receive data from it, Universal Robots' (UR) Real-Time Data Exchange (RTDE) interface, which is available by default on UR's robots, is used. The RTDE is a low-level communication protocol that operates over a standard TCP/IP connection, enabling external applications to exchange data and control signals with the UR controller at a high frequency [6].

The communication protocol itself was implemented using the open-source `ur_rtde` C++ library developed in cooperation with SDU Robotics [7]. The library abstracts the complexities of the binary RTDE protocol, providing a user-friendly object-oriented wrapper, while logically splitting the interface in 3 parts;

control, receive and I/O. This allows the user to control movement and data collection separately and at the same time. For this project, the I/O interface was not used as the gripper is controlled on a dedicated separate TCP/IP connection.

The `ur_rtde` library provides C++ movement methods, such as `moveJ` and `moveL`, that closely correspond to those found in the UR5 teach pendant. Since the movement path for the throwing sequence was pre-computed, the main control loop utilizes the non-blocking `speedJ` control command for execution. This command allows the user to continuously input a target joint speed vector, $\dot{\mathbf{q}}_{\text{target}}$, at the control loop frequency (125 Hz for the UR5). The command also requires a specified acceleration limit to reach the target speeds, and a Δt (time step) parameter to define the duration of the current control interval. By looping through the pre-computed joint speed sequence, $\dot{\mathbf{q}}$, at the 125 Hz rate, the desired throwing movement is executed.

Movements other than the throwing sequence are also required, these movements use `moveJ` as to limit the risk of hitting a joint boundary, while not having to map the movement fully beforehand. `moveJ` allows the user to input a target joint position, a desired speed and acceleration. The move is then executed by interpolating linearly in joint space, which ensures a controlled path and reduces the range of motion needed at each joint. Theoretically, joint boundaries should never be violated using this movement type, provided both the start and end configurations are within the joint limits. This movement type is used for the static ball pickup sequence, for moving the robot to the throwing sequence start position, and for returning to a defined home configuration after the throw.

5 Data Collection

In order to iteratively improve the performance and understand the behavior of the system, a robust data collection mechanism was implemented. This process is critical for validating the calculated path against the robot's actual motion, particularly during the high-speed throwing sequence.

For each calculated throw, a comma-separated values (CSV) file is generated. The file is initially populated with the crucial input parameters and pre-computed trajectory data, including:

- Target and release points
- Ballistic parameters (e.g., pitch, yaw, release velocity)
- Status code
- The complete pre-computed sequences of joint positions ($\mathbf{q}$) and velocities ($\dot{\mathbf{q}}$)

During the execution of the movement, the actual joint positions and velocities achieved by the UR5 are requested from the robot controller using the RTDE Receive Interface. These real-time values are simultaneously logged to the same CSV file. This combined dataset allows for a direct comparative analysis between the modeled throw (MATLAB output) and the executed throw (RTDE feedback).

To physically see where the ball ended up, a Velcro target is used. This target is placed on a white background, to distinguish between it and the table easier, and a 3d printed set of rings were placed at the centre. These rings are needed as the Velcro of the target became less sticky, allowing bounces and more movement than we could measure effectively. The rings start at 120mm inner diameter and go all the way down to 45mm. These rings go down in diameter by 1 cm for each ring, the only exception being the "ideal target" ring. This ideal was decided in the beginning of the project, needing to hit within 2cm of the centre, resulting in a circle with a 74mm diameter. This ring was made slightly thicker, making the step larger, but also more stable as to not move this ring. The way these rings are used is by adding them one by one into the previous, allowing for testing of smaller targets and higher precision.

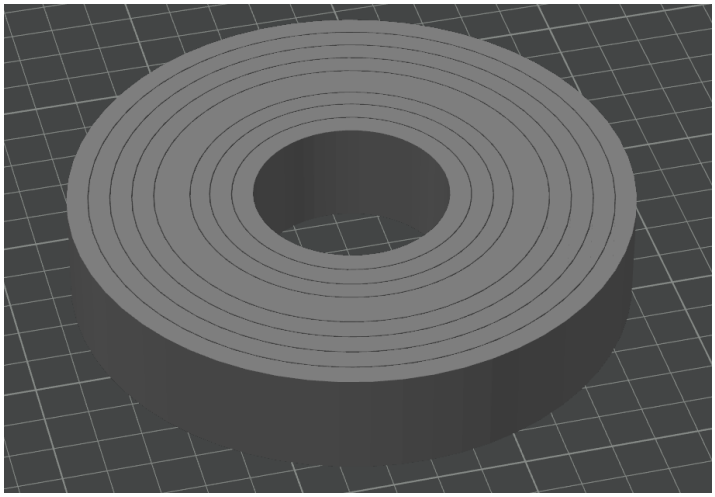


Figure 5.1: 3D printed rings used for precision testing

6 Integration

The final program takes all the separate parts, encapsulated in classes, and merges it into a state machine that calls the various needed classes and functions. When run, both the connections to the gripper and UR5 is established, and a menu prompts the user with 5 options:

- Calculate a trajectory
- Throw the ball
- Calibrate/Settings
- Manual control
- Exit program

To calculate the trajectory, the vision first finds a circle and the centre of said circle. This is the target. The MATLAB script is then called, calculating the movement sequence and its data points. All the data is then returned, sorted and stored, allowing the user to initiate the throw. Throwing the ball consists of 3 steps; picking up the ball, throwing it, and prompting the user to save the real-time data from the throw.

Because there was a possibility that different robot cells had to be used, a calibration menu is built in. This allows the user to calibrate a new table, or select from a list of earlier calibrations.

Lastly manual control is available, allowing the user to execute smaller movements of both gripper and robot. All the options can be seen in the flowchart 6.1

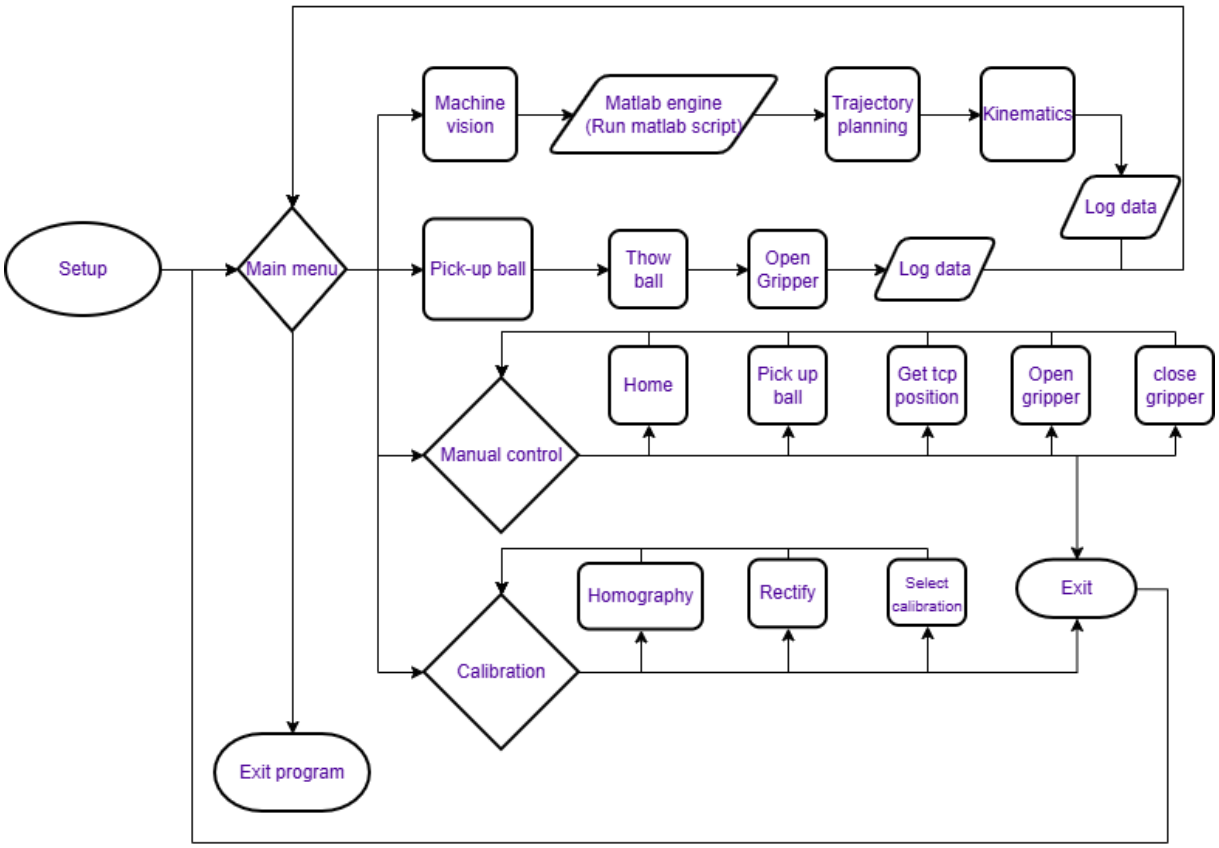


Figure 6.1: Flow chart of the finished system

7 Results

Testing the system and its components can be done in many ways. Because so many parameters can be tweaked, it was important to test many of them to determine their individual effects on the system as a whole. The results from all tests will be discussed in chapter 8.

Initial testing was done using simulated throws, by running the trajectory planner, including the ball ballistics and robot path planning, with a range of release point and targets. The very first test was done to evaluate the impact of using a weighted Jacobian to influence which joints were most active during the throw movement. Table 7.1 shows the results from 5600 simulated throws across a range of release points and targets, the distribution of which can be seen in figure 7.1a.

Table 7.1: Simulation Test: Weighted vs Unweighted Jacobian

Status	**Description**	**Unweighted**	**Weighted**	
21	IK release position off by > 5 mm	1424	1424	
23	Joint limits exceeded in lead-up	827	848	(+21)
24	TCP limits exceeded	156	156	
26	Release position too close to target	144	144	
100000	Perfect Hit ($<$ 1mm offset, clean)	78	91	(+13)
100100	$<$ 1mm Offset + Follow-through Clipped	22	0	(-22)
1X0000	1-5mm Offset (clean)	97	135	(+38)
1X0100	1-5mm Offset + Follow-through Clipped	52	2	(-50)
Targets hit (of 28 possible)		20	20	
Settings: Acceleration 5 rad/s^2, Min. distance 20 cm				

Using a weighted Jacobian, which sets the cost of using Wrist 2 at twice the amount of all other joints, impacts both the lead-up and follow-through sequence. During lead-up, it becomes more common that a joint limit is exceeded, likely because some joints have to move more to compensate for the minimized movement in Wrist 2. However, when a path is found, it is more likely to be clean, i.e. not experiencing any clipping in the follow-through sequence. Both tests were able to reach the same amount of targets, with the targets near the side of the table being the unreachable ones.

Further simulation tests were done to evaluate the impact of the model's acceleration. Since all robot path planning is calculated based on the model's constant acceleration, changing this one parameter will likely dictate which targets are reachable. Table 7.2 shows the impact of 3 different accelerations when enforcing a minimum horizontal distance of 20cm between release and target point. Additionally, table 7.3 shows testing with a minimum distance of 35cm. The distribution of release points and targets for these tests can be seen in figure 7.1a and 7.1b respectively.

The tests show similar results, where increasing the acceleration decreases the risk of hitting a joint limit. However, the increase in acceleration also causes more throws to exceed the TCP limits. Additionally, it is seen that more viable paths are found and that more targets can indeed be reached.

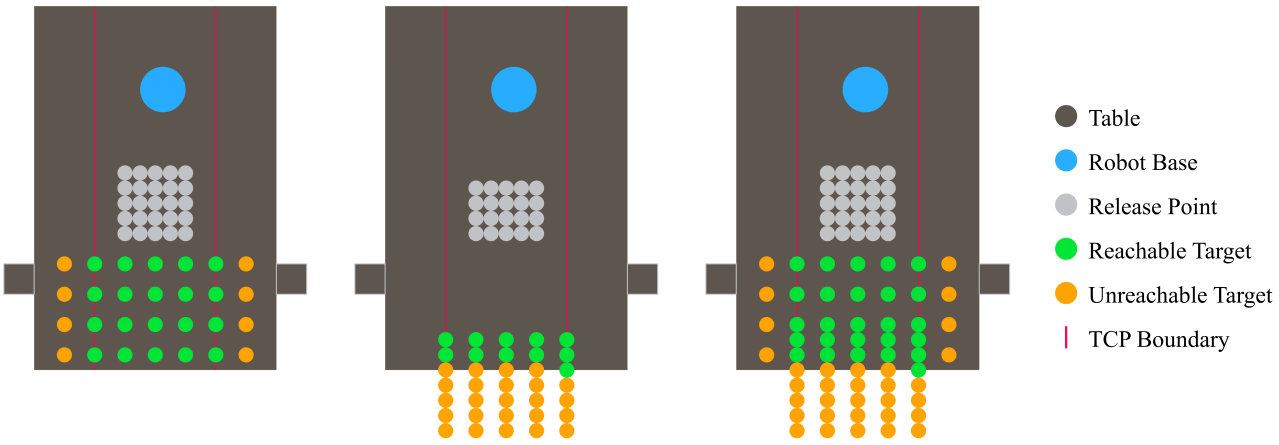
In addition to the simulated test throws, a series of real-world throws were also conducted. Since the model and robot can have different accelerations applied to them, the relationship between the two could also impact the robot's precision. Figure 7.2 shows the Mean Absolute Error in rad/s across different relative accelerations.

Table 7.2: Simulation Test: Acceleration

Status	**Description**	**Acc.** 4 rad/s^2	**Acc.** 5 rad/s^2		**Acc.** 6 rad/s^2	
21	IK release position off by $>$ 5 mm	1423	1424	(+1)	1424	
23	Joint limits exceeded in lead-up	960	848	(-112)	741	(-107)
24	TCP limits exceeded	120	156	(+36)	182	(+26)
26	Release position too close to target	144	144		144	
100000	Perfect Hit ($<$ 1mm offset, clean)	59	91	(+32)	127	(+36)
100100	$<$ 1mm Offset + Follow-through Clipped	0	0		0	
1X0000	1-5mm Offset (clean)	92	135	(+43)	180	(+45)
1X0100	1-5mm Offset + Follow-through Clipped	2	2		2	
Targets hit (of 28 possible)		17	20	(+3)	20	
Settings: Min. distance 20 cm, Weighted Jacobian						

Table 7.3: Simulation Test: Acceleration at Distance > 35cm

Status	**Description**	**Acc.** 5 rad/s^2	**Acc.** 6 rad/s^2		**Acc.** 7 rad/s^2	
21	IK release position off by $>$ 5 mm	652	652		653	(+1)
23	Joint limits exceeded in lead-up	2052	1945	(-107)	1818	(-127)
24	TCP limits exceeded	40	95	(+55)	181	(+86)
26	Release position too close to target	0	0		0	
100000	Perfect Hit ($<$ 1mm offset, clean)	31	58	(+27)	75	(+17)
100100	$<$ 1mm Offset + Follow-through Clipped	0	0		0	
1X0000	1-5mm Offset (clean)	25	49	(+24)	71	(+22)
1X0100	1-5mm Offset + Follow-through Clipped	0	1	(+1)	2	(+1)
Targets hit (of 35 possible)		11	11		12	(+1)
Settings: Min. distance 35 cm, Weighted Jacobian						



(a) Initial Testing (b) Distance Testing (c) Combined Overview

Figure 7.1: Overhead Visualization of Experimental Workspace and Target Distribution

The model uses a constant acceleration of $5 rad/s^2$ while the robots acceleration is varied relative the model's.

It is seen that a slight increase in the robots acceleration relative to the model's, generally decreases the absolute mean error. For all further tests, the robots acceleration will be set to 25% more than the model's.

Table 7.4 shows the results of 100 throws across different target objects and release timings. The two targets used were the dart board and the plastic ring with a diameter similar to the bullseye of the dart board.

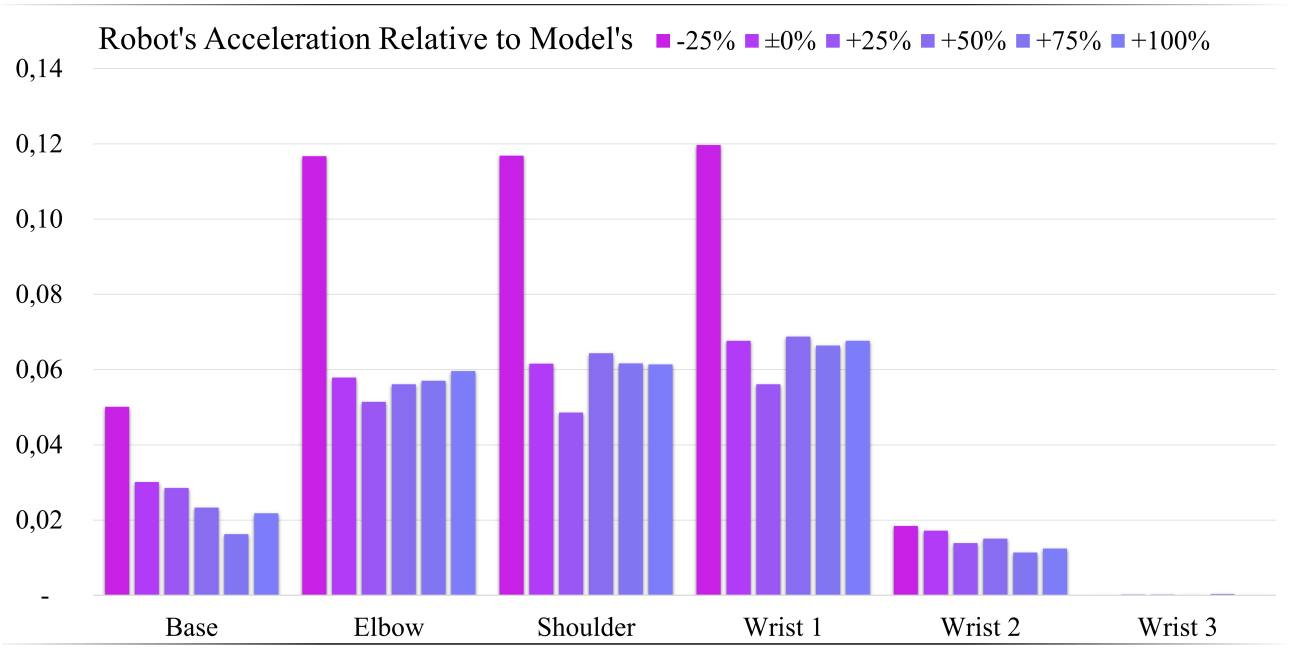


Figure 7.2: Mean Absolute Velocity Error by Joint in rad/s. Model Acceleration 5 rad/s^2

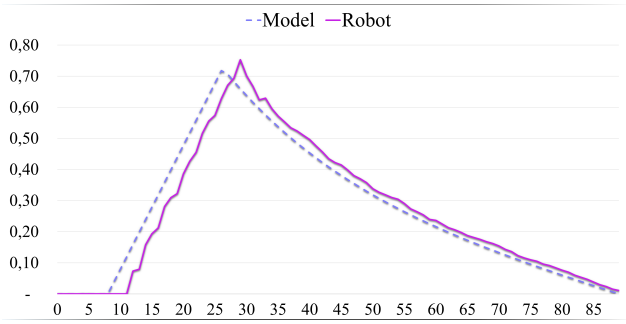
The release offset determines when the gripper is called to release the ball, relative to the time step in which the ball should lose contact with the robot, i.e. if the ball should be in the air at time step 35, the gripper is called at time step 33 when the offset is 2. The results are segmented into misses, boundary hits and clean hits. A clean hit is defined such that the entire ball must make it inside the bullseye or ring. A boundary hit is counted when the ball is partially inside the bullseye or hits the ring itself. A miss is noted if the ball does not meet one of the two hit requirements.

Table 7.4: Physical Test: Accuracy

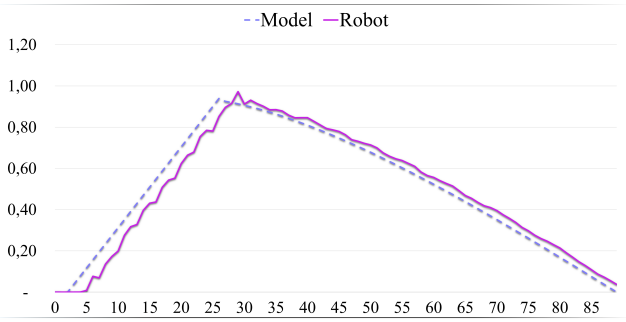
Target	**Release Offset**	**Misses**	**Boundary Hits**	**Clean Hits**	**Success Rate**
Dartboard	2 Time Steps, 16ms	9	9	7	28%
Plastic Ring	2 Time Steps, 16ms	10	5	10	40%
Plastic Ring	5 Time Steps, 40ms	9	7	9	36%
Plastic Ring	6 Time Steps, 48ms	5	3	17	68%

The first three setups show relatively similar results, with only the last setup being significantly different. The setup using a 48ms release offset resulted in only 20% misses compared to around 40% for the other three. Additionally, the last test showed a considerable increase in clean hits, with only 15% of total hits being boundary hits.

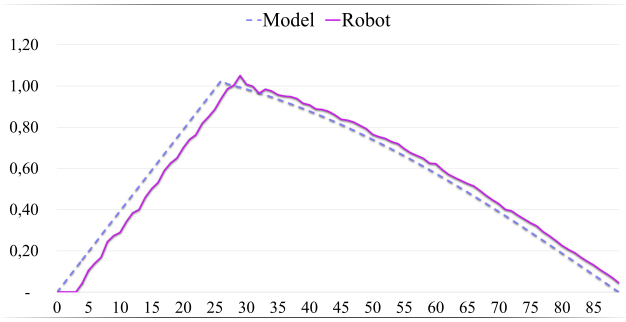
With the data from both the model and robot, it is possible to compare their velocities over time during a throw. Figure 7.3 shows the velocity of both robot and model for the 4 most active joints during a throwing sequence.



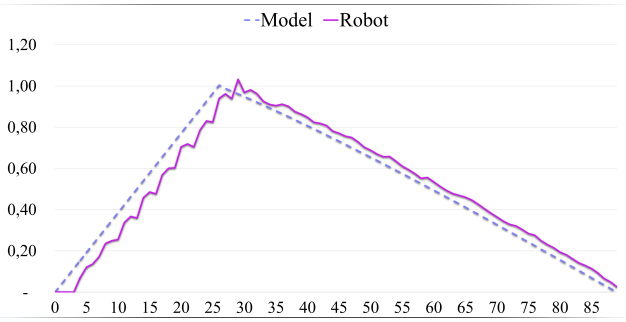
(a) Base



(b) Shoulder



(c) Elbow



(d) Wrist 1

Figure 7.3: Comparison of Velocity Over Time Between Model and Robot in rad/s

8 Discussion

8.1 Machine Vision

8.1.1 Hough Transform

Instead of `cv::HoughCircles`, other methods like searching for specific patterns within the defined target or looking at the colors of the target were considered. However, since the target and many other objects within the scene are circular, the Hough transform is both more versatile and more robust. Additionally, it is very easy to implement with OpenCV compared to any specific algorithm hardcoded to this scenario.

8.1.2 Computing Homography

Another, perhaps more robust, way of selecting points to compute the homography would be to find the circular holes in the table using `Vision::findCircularObject` and use that as the points for the homography as they are also known positions and would be more precise than having the user click on the pixels that correspond to the table corners. However, this was not implemented, as the original solution worked as intended.

8.2 Trajectory Planning

Trajectory generation for the ball and robot involves various methods with distinct trade-offs. For this project, methods were selected based on their impact on the primary goal. Because trajectory planning involves many interdependent parameters, it functions as an optimization problem where a "perfect" solution is elusive. Consequently, the approach to selecting variables and calculation methods varied; some were chosen through rigorous testing, while others relied on hypotheses and limited validation.

8.2.1 Ball Trajectory

The ball's trajectory was optimized to meet specific project constraints. To qualify as a "throw," the object must travel horizontally with an upward trajectory at release. Thus, calculations enforce a minimum pitch of 22.5° ($\pi/8$ rad). This constraint is vital because the secondary objective—minimizing velocity—would otherwise favor a negative or near-zero pitch, given that the release point is higher than the target. Lower velocity is preferred as it reduces the required acceleration space, limiting the robot's range of motion and ensuring it stays within physical boundaries.

8.2.2 Robot Path Planning

As the joint limits are more restricting than the Cartesian boundaries, planning in joint-space seems optimal. While joint-space planning offers total control over joint positions, it complicates generating specific Cartesian movements. The primary objective is ensuring the Tool Center Point (TCP) reaches the release point with the correct velocity and direction. To simplify this, the path is split into two section:

Lead-up Trajectory: This phase aims to reach the release point at the target speed while respecting joint limits. By calculating this in joint space, the range of motion is optimized based on acceleration capabilities.

However, the precise Cartesian velocity and direction are only guaranteed at the exact moment of release; prior points may have arbitrary vectors.

Follow-through: To account for mechanical release timing errors (e.g., the ball not losing contact instantly), the TCP continues along the ball's trajectory for a set duration. This phase is calculated in Cartesian space to enforce a linear direction. Simultaneously, the robot decelerates to remain within safety limits. This deceleration also provides a buffer: if the ball is released late, its proximity to the target is compensated for by a lower release velocity, maintaining some degree of accuracy.

8.2.3 Robot-Table Calibration

The system developed in this project integrates multiple components that enable the robot to perceive its environment, plan feasible trajectories, and execute accurate motions within a calibrated workspace. The system consists of five core modules: machine vision, robot–table calibration, trajectory planning, kinematics, and robot control, which operate sequentially and are closely integrated.

The machine vision module captures images of the table surface and detects the center of the target. The detected position is initially expressed in the camera frame and subsequently transformed into world-frame coordinates for use in the planning and control stages.

Trajectory planning computes a feasible motion from the robot's current configuration to the target location while respecting kinematic and workspace constraints. Kinematic computations convert the planned end-effector motion into joint configurations using inverse kinematics, with forward kinematics used for verification.

The robot control module executes the planned trajectory by commanding the robot joints to follow the desired motion profile. Together, these modules form an integrated system that enables reliable operation within the table workspace. An overview of the experimental setup is shown in Figure 1.2.

8.2.4 MATLAB Communication

In the final program MATLAB Engine was used, but other options were explored. Alternatives include but are not limited to MATLAB MEX functions and CSV communication. CSV is the crudest version of this communication, as it required MATLAB to already be running and checking the repeatedly CSV file for changes. The code did work, but having two programs running defeated the point of integrating MATLAB to have one seamless system, so it was decided to move away from CSV. Some of the code was repurposed for the storing of data, as mentioned in chapter 5.

The other options explored are native to MATLAB. MEX is a way of recompiling a MATLAB script into something that can be directly integrated into, among others, C++ [8]. It works by compiling both the script and the C++ code in MATLAB and running it in that environment. That means that data can be passed back and forth directly in memory and would in theory be faster than MATLAB engine. Downsides do however include limitations on toolboxes that were needed, and the fact that running it in MATLAB could introduce more problems with communication with the robot and camera.

MATLAB engine works by starting the script as a process in the background, passing the data by value through the API, running it through the script, and returning it by value through the API to the C++ code [9]. This is in theory slower, and with more data than needed for this project, it could be an issue having multiple copies of the same data. However, using it allowed running the C++ code natively instead of in MATLAB and was easier to compile with libraries. Additionally, it allowed the use of toolboxes like the Robotics System Toolbox, while still integrating MATLAB directly with the C++ code, and therefore MATLAB engine was the preferred option in this project.

8.3 Results

The results from the testing phase gives an indication as to how each parameter impacts the system. Initial testing was done by simulating throws using varied parameters to find a suitable setup for the real-world testing.

8.3.1 Weighted Jacobian

Using a weighted Jacobian forces some joints to be more active. The implementation of this was done to limit the use of Wrist 2, which is highly restricted by the safety limits. This causes an increased usage of the other joints which resulted in the joint limits being exceeded more often during the lead-up path. However, it was determined that the increase in non-clipped follow-through paths out-ways the negative impact during lead-up, and thus the weighted Jacobian was used for all further testing.

8.3.2 Acceleration

It is natural that the robot cannot accelerate at an infinite rate, but finding which acceleration is possible with limited error margins proved to be difficult. Testing the impact of acceleration on the kinematic model was the first step to find a suitable level. It was found that an acceleration of $5 rad/s^2$ seemed to be a good middle ground, as it allows for hits at almost all tested target points. Increasing the acceleration by 20% to $6 rad/s^2$ only added one additional reachable target. Keeping the acceleration relatively low is desirable as to keep the robot from having to do aggressive movements.

As the robot path planning uses a static acceleration for the lead-up sequence, it is imperative that the robot is able to hit this acceleration profile as well. The movement control of the robot during the throw sequence is done by providing target speeds to the joints at each time step. The acceleration allowed to hit those targets must also be specified. Naturally, an acceleration lower than that of the model means it is impossible for the robot to hit the target speed at any point. Additionally, using an acceleration much higher than the model may lead to the robot stuttering as it will only need to accelerate at the start of each time step. A balance was found at a robot acceleration 25% higher than the model, this provided the lowest mean absolute velocity error, meaning the robot's actual velocity was the closest to the model's.

8.3.3 Release Timing

It was quickly noted that there is a delay between calling the gripper to release the ball, and the ball physically being free from the robot. Determining this delay precisely was nearly impossible without any sensor feedback showing when contact with the ball was lost. Two approaches was used to estimate the delay, first, the gripper and terminal output was filmed in slow motion at 480 frames per second, and secondly, testing was done using varied offsets. Using the slow motion video it was estimated that the delay was 20-25 frames translating to 40-50ms, which was backed by the testing which showed the best results at an offset of 48ms.

8.3.4 Velocity Profile

The method of splitting the path in two sections leads to a velocity profile in the shape of a triangle as seen in figure 7.3. In theory the path could have been split in a much larger number of sections, which would allow for more complex velocity profiles, but would also increase the complexity of the kinematics and mathematics needed substantially. The theorized benefit of using more sections is to allow for a differentiated acceleration throughout the sequence. This could help minimize the effects of jerk experienced as each joint starts moving.

Had the time frame allowed, it would have been interesting to further test this hypothesis and try different profiles.

8.3.5 Accuracy

The accuracy shown by the final system does not fully meet the expectations defined in section 1.2. The best results achieved by the system, seen in table 7.4, managed only a 68% success rate compared to the desired 90% consistency. Of the remaining 32% of throws in that test, 3 of 8 were boundary hits, meaning they were very close to being within the tolerances. Increasing the consistency likely comes down to tweaking some of the parameters discussed in the earlier sections.

9 Conclusion

The objective of this project was to design and implement a robotic system capable of detecting a target on a table and executing a throwing motion toward the detected target using a UR5 robot. This required the integration of calibration, machine vision, trajectory planning, and robot control within a unified framework.

A robot–table calibration based on an SVD-based least-squares method was implemented, enabling transformation between world-frame and robot-frame coordinates with sufficient accuracy to support vision and motion planning. A vision-based target detection approach using a fixed camera and homography mapping allowed target positions on the table plane to be expressed in world coordinates and used as input to the trajectory planning module. Throwing trajectories were generated using a simplified projectile motion model and executed using predefined lead-up, release, and follow-through phases within the robot's kinematic and safety constraints.

The implemented system successfully demonstrated autonomous target detection and execution of corresponding throwing motions, fulfilling the primary functional objectives of the project. However, the specified accuracy requirement of consistently hitting within 2 cm of the target center was not fully achieved. Despite this, the project resulted in a complete end-to-end robotic throwing system that satisfies the overall project objectives and provides a solid foundation for further refinement.

References

[1] Lecture notes by Thorbjørn Mosekjær Iversen, *Calibration of the Vision Setup in Robolab*, September 2023, Available on ItsLearning

[2] Christoffer Sloth, *Lecture Notes on Robot to Table Calibration*, October 20, 2019, Available on ItsLearning

[3] SCHUNK GmbH and Co. KG., *GCL _ manual(oversat).PDF*, Version: 01.00 15/08/2017en, Document number: 1005737

[4] MathWorks, *Robotics System Toolbox*, Accessed 16.12.2025.
Available at: https://se.mathworks.com/products/robotics.html

[5] Inigo Iturrate, *UR5_kinematics.zip*, November 2025, Available on ItsLearning

[6] Universal Robots A/S, *Real-Time Data Exchange (RTDE) Guide*, Accessed 16.12.2025
Available at: https://docs.universal-robots.com/tutorials/communication-protocol-tutorials/rtde-guide.html

[7] A. P. Lindvig, I. Iturrate, U. Kindler, and C. Sloth, "ur_rtde: An Interface for Controlling Universal Robots (UR) using the Real-Time Data Exchange (RTDE)," in *2025 IEEE/SICE International Symposium on System Integration (SII)*, 2025, pp. 1118–1123. DOI: https://doi.org/10.1109/SII59315.2025.10871000

[8] The MathWorks Inc. *Build MEX functions and engine or MAT file applications* Accessed 16.12.2025 Available at: https://se.mathworks.com/help/matlab/ref/mex.html

[9] The MathWorks Inc. *Call MATLAB from C++* Accessed 16.12.2025 Available at: https://se.mathworks.com/help/matlab/calling-matlab-engine-from-cpp-programs.html

A Distribution of tasks

B Code

C Video Showcase

A video of the system running is available at:
https://youtu.be/TekeDc8nFaY