Semester project on signals in robot systems

BEng in Robot Systems

3. Semester

Authors (Group 3)

Name	Username	Birthday
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 12, 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. Possibilities 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 intend 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 far the average throw is from the bullseye.

1.2 Problem Definition

Make 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 modeled trajectory

The solution is expected to consistently hit within the visual bullseye of the Velcro dart board, as seen in figure **??**. 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. Additionally, the solution should require no human intervention once started, apart from placing the ball in a designated pickup area at the start of each throw.

1.3 Constraints & Limitations

1.4 System Overview

2 Machine Vision

2.1 Calibration

2.2 Homography

2.3 Finding circular objects

2.4 Implementation with OpenCV

3 Trajectory Planning

3.1 Physics

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 matched 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

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) ref. 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

The first step in the process is determining what joint position will provide the required release position in Cartesian space while staying within the joint limits. This is done using the generalized inverse kinematics solver from RST. This solver uses an optimization algorithm to find a solution that best satisfies a list of requirements. Four things are provided to the solver; joint limits, desired TCP position, desired TCP orientation and desired joint configuration.

Firstly, the joint limits are found directly in the teach pendant of the UR5 robot. An offset is applied to these limits to ensure that the solution is not close to a joint boundary, which would highly limit any movement of the joint in the lead-up and follow-through paths. The offset limits are converted from degrees to radians and input to the solver using a weight of 500, which determines its cost in the solvers cost-limiting optimization algorithm.

Secondly, the release position is converted from world frame to base frame using the transformation matrix found during calibration and input to the solver with a weight of 50.

Thirdly, the desired TCP orientation is found based on the yaw and pitch found during the ball trajectory planning. The angles are converted into a normalized vector in world frame. A rotation matrix is created such that the Y-axis points in the opposite direction of the vector. The Z-axis is set to point as much downwards as possible in the world frame while still remaining orthogonal to the Y-axis. The X-axis is set to be orthogonal to both the Y and Z-axis, completing the right-handed frame. The rotation matrix is then converted from world frame to base frame. The main objective of this orientation is to ensure that the gripper is not in the way of the ball after its release. The completed rotation matrix is input to the solver with a weight of 1.

Lastly, the solver is given a desired joint pose. This pose is set to the center of each joints' allowed movement range in order to target solutions that give the maximum range of motion in both direction.

If the solver does not managed to find a pose that satisfies the inputs, an error code is received and stored. Otherwise, the solver's solution, defined as $\mathbf{q}_{\text{release}}$, is checked to ensure it is within tolerance. This check is done by finding the distance from the desired TCP position to the computed TCP position, if the computed position is more than 5 mm off, an error code is set.

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 $\mathbf{v}_{\text{release}} = vel \cdot \mathbf{dir}$.Firstly, the velocity Jacobian ($\mathbf{J}_{\text{vel}}$) is found for the joint pose. This 6×6 matrix is derived from the robot's kinematic model using the Matlab function provided by Inigo Iturrate. 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 (the first 3 rows) 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:

$$\mathbf{J}_{\text{weighted}} = \mathbf{W}^{-1}\mathbf{J}_{\text{vel}}^{T}(\mathbf{J}_{\text{vel}}\mathbf{W}^{-1}\mathbf{J}_{\text{vel}}^{T})^{-1} \tag{3.1}$$

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

$$\dot{\mathbf{q}}_{\text{release}} = \mathbf{J}_{\text{weighted}}\mathbf{v}_{\text{release}} \tag{3.2}$$

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:

$$\text{time}_\mathbf{j} = \frac{|\dot{\mathbf{q}}_{\text{release},j}|}{a} \tag{3.3}$$

$$\text{steps}_\mathbf{j} = \lceil \text{time}_\mathbf{j} \cdot f \rceil \tag{3.4}$$

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):

$$\dot{\mathbf{q}}_{i,j} = \dot{\mathbf{q}}_{\text{release},j} \cdot \max\left(\left(1 - \frac{i}{\text{steps}_\mathbf{j}}\right), 0\right) \tag{3.5}$$

$$\mathbf{q}_{i,j} = \mathbf{q}_{i-1,j} - \dot{\mathbf{q}}_{i,j} \cdot \Delta t \tag{3.6}$$

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.

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 ($\mathbf{J}_{\text{weighted,i}}$). The Jacobian is again found using the kinematics function and transformed into a weighted pseudo-inverse with only the velocity parameters. With the joint velocity, the joint position can be updated. The iteration index i runs from $i = 1$ to $\text{steps}_{\text{follow}}$

$$\text{steps}_{\text{follow}} = round(0.5 \cdot f) \tag{3.7}$$

$$\mathbf{v}_{\text{i}} = (1 - \frac{i}{\text{steps}_{\text{follow}}}) \cdot \mathbf{v}_{\text{release}} \tag{3.8}$$

$$\dot{\mathbf{q}}_{\text{i}} = \mathbf{J}_{\text{weighted,i}} * \mathbf{v}_{\text{i}} \tag{3.9}$$

$$\mathbf{q}_{\text{i}} = \mathbf{q}_{\text{i-1}} + \dot{\mathbf{q}}_{\text{i}} \cdot \Delta t \tag{3.10}$$

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.

Status codes

Throughout the path generation the safety checks will provide status codes or error codes if anything is outside the tolerances or a limit is enforced. Two types of codes can be generated, either the safety check forces the path generation to stop completely and gives an error code, or it is accepted, noted in the status code, and the generation continues. The errors that stop the generation are mainly errors that occur if the path is outside a safety limit, either in joint or Cartesian space. These errors are all noted as two digit numbers with the left digit being 2. E.g. if the path is outside TCP limits, the error code will show 24.

The status code is updated throughout the calculations whenever the calculation shows some imperfection. If a path is generated with no imperfections it will show 100000. Each digit after the left-most 1 indicates a different imperfection. The digit in the second position from the left shows how far the calculated release position is from the desired release position in the world frame and is noted in mm. E.g. a status of 130000 means a path was generated but the TCP will be off by 3mm at release. The full list of status codes is as follows:

- 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

4 Robot Control

4.1 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, was 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 ref.

The communication protocol itself was implemented using the open-source `ur_rtde` C++ library developed by SDU Robotics ref. 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 robotic 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).

noget om plastik ringene?

6 Evaluation

7 Discussion

7.1 Trajectory Planning

Generating a trajectory for the ball and subsequently a path for the robot can be done in a multitude of ways, each with pros and cons. Therefore, one should consider what the objective is and define the problem in detail before choosing a method. The methods chosen for this project were all decided based on their expected impact on the project goal.

As the complete trajectory planning includes a substantial amount of parameters that can be tweaked endlessly, it essentially becomes an optimization problem. Since each parameter change will affect how all other parameters act, finding the perfect solution for any throw is almost impossible. Due to this fact, the approach when choosing static variables or calculation methods, differed based on the expected impact of that variable. For some calculations rigorous testing was done to find the impact of change in those variables, while for others simple hypothesis and limited testing was applied to make the decision.

7.1.1 Ball Trajectory

The ball's trajectory can be optimized to achieve different goals. The problem definition dictated the decisions as to how these calculations were done. In the spirit of the project, *throwing* an object was defined such that the object must travel some distance horizontally and must have an upwards trajectory at the release point. In order to achieve this, the ball trajectory calculation enforces a minimum pitch of 22.5 degrees ($\pi/8$ *radians*). This enforcement is important, because the secondary goal of the calculation is to find the trajectory which requires the lowest necessary velocity. A low velocity is helpful when determining the robots path, as less space is needed to accelerate, thus limiting the range of motion and increasing the chance of staying within limits.

However, since the TCP cannot get near the table, where the target is placed, all throws will have a release point higher than the target point. The lowest velocity necessary for such a throw is likely found with a negative or near 0 pitch, which does not satisfy the project definition of a throw. And thus the arbitrary enforcement of 22.5 degrees was implemented.

7.1.2 Robot Path Planning

As the robot is limited in both Cartesian and joint space, planning the path in either space leads to an unknown degree of complications in the other. As the joint restriction are generally harsher than the Cartesian boundaries, it was natural to do the path planning in joint space. This allows for full control of the joints positions at all times, however, it also limits the ability to generate specific movements in Cartesian space. The most important task of the path planning is to ensure that the TCP hits the release point in Cartesian space while moving in the direction of the ball's

calculation trajectory with the correct velocity. Splitting the path planning in two sections makes this task much easier to achieve, as it is now possible to define the release point as either a start of end position in the trajectory, rather than some point along the path. With the path split in two, each section can now aim to achieve different goals.

The lead-up trajectory's goal is to hit the release point with the correct speed, this must be done while staying within joint limits. Calculating this part in joint space means it is possible to limit the range of motion needed for each joint based on its acceleration. However, that also means the correct velocity and direction in Cartesian space can only be ensured at the exact point of release. Any point before that may have any random direction and velocity at the TCP.

Since some error margin in the release timing must be expected, i.e. it might not be possible to have the ball lose contact with the gripper in that exact moment in time, it seemed optimal to have the TCP move in the direction of the ball trajectory for some amount of time. This becomes the goal of the follow-through section of the path. Determining this sequence in Cartesian space, allows the calculations to enforce a linear Cartesian direction of the TCP. During this movement, the robot also decelerates, which achieves two things. Firstly, the robot needs to slow down before hitting its physical limits and safety restriction. Secondly, the deceleration may counter the error margin in release timing. If the ball is released slightly too late, it will still be traveling in the correct direction towards the target, and though it is released closer to the target than expected, it is also released with a lower velocity.

8 Conclusion

9 References

A Distribution of tasks

B Code