236754 - Project in Intelligent Systems

Robotic Hand for Object Tracing

Yonatan Reshef
Guy Sudai
Maxim Wainer

Source code can be found at: https://github.com/GeckoWarrior/RoboticArm.git

Introduction

Robotic arms are increasingly finding their place in medical environments, assisting surgeons with tasks that require precision, stability, and endurance beyond human capability. From automated tool handling to steady illumination, such systems can enhance both accuracy and efficiency in critical procedures.

In this project, we present an end-to-end system for vision-based object tracking with the Universal Robots UR5e, equipped with a mounted camera. The arm continuously follows and centers a target in its field of view, enabling applications such as maintaining a spotlight on surgical tools or keeping the surgeon's working area perfectly illuminated. Our approach combines computer vision, real-time control, and robotic kinematics to deliver smooth, responsive tracking suitable for demanding medical contexts.

Keywords: Robotic arm, UR5e, object tracking, computer vision, camera-based control, real-time tracking, automation.

Problem Definition

Given a set of targets S_T located within the camera's field of view and a robotic manipulator R (UR5e) equipped with a mounted camera, choose an object T from S_T and continuously adjust R's end-effector pose such that the projection of T remains centered in the image plane.

We divide the problem into three main components:

1. Object Detection: Given the camera image at time t, detect and localize all candidate objects $\{T_1,..., T_n\}$. This stage ensures robustness in cluttered scenes and provides the input set for subsequent selection and tracking.

2. Visual Tracking: From the detected set, select a target object T_k and determine its pixel coordinates (u, v) relative to the image center (u_0, v_0). The objective is to provide accurate, real-time localization of T_k, while handling noise, and partial occlusions.

3. Manipulator Control: Instead of directly commanding poses, we compute end-effector velocities $v = [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z]$ that reduce the distance over time.

This formulation ensures smooth and bounded end-effector motion, while making the system modular: the detection and tracking components can be reused across different tasks, and the velocity-based control can be adapted to various medical applications, such as stabilizing a spotlight on surgical tools or maintaining a consistent view of the operative field.

Problem Overview

Traditional vision-based control of robotic arms often relies on discrete pose corrections. The system estimates the current distance between the target's location and the desired position in the image, then commands the manipulator to move directly to a new pose. While this approach is straightforward, it has several limitations. First, discrete jumps in pose can result in jerky motion of the arm, which is undesirable in sensitive applications such as surgical assistance. Second, repeated repositioning introduces delays, since the robot must decelerate and re-accelerate at each correction step. Finally, abrupt changes in orientation or velocity may increase mechanical stress on the manipulator and reduce overall stability.

A velocity-based control strategy offers a more robust alternative. Instead of commanding the arm to "jump" between poses, the system computes continuous linear and angular velocities that smoothly reduce the error over time. This allows the manipulator to adjust its motion gradually, leading to steadier tracking, less mechanical wear, and more reliable performance. The principle is similar to how a human adjusts their hand when holding a laser pointer on a moving object: rather than moving in steps, the hand continuously corrects the trajectory to keep the beam aligned.

Challenges - Detection System

Detection of Objects

We initially experimented with deep learning–based object detection models for the detection of objects. We tried using the YoloV10 model, and while it offered strong generalization, it proved too slow for real-time use and trying to use the smaller YoloV10n model resulted in unconsistent detection. As a result, we shifted to more classical methods, such as computing color gradients and edge responses to identify object contours, from which bounding boxes could be extracted manually. Although less sophisticated, and requiring a preset for each type of object, this approach provided the consistency and speed needed for continuous operation.

Calculating Position Relative to Camera

We needed to calculate the object's (x,y,z) position, relative to the camera. Since we were not able to access all the cameras on the EYES module, we had to find an alternative method for estimating the object's depth. To achieve this, we preformed a depth estimation by apparent size based on the pinhole camera model. To do this we required knowledge of the camera's intrinsic parameters, represented by the calibration matrix K.

To obtain these parameters, we performed a camera calibration procedure using multiple images of a 9×6 checkerboard pattern captured from different angles. This allowed us to accurately estimate the intrinsic characteristics of the camera, and calculate the depth of the object (in meters).

In order to calculate the object's x and y coordinates (in relation to the camera in meters) , we used its already calculated depth (z value) and the calibration matrix K that we found, to preform a reverse projection transform, which makes it possible to infer 3D information from the object's 2D image size, and get the x and y coordinates.

Tracking the Object Through Frames

The tracking state involves not only recognizing the object in the current frame but also analyzing its motion dynamics over time. By understanding the trajectory, velocity, and direction of movement of the tracked object, the system can generate an informed prediction of its future position. This prediction is then used to assign a bounding box that is aligned with the anticipated location of the object, allowing the tracker to maintain a consistent association with the same target even in the presence of noise, temporary occlusions, or sudden changes in appearance.

Challenges - Control System

Multiple Coordinate Systems
In robotic manipulation tasks involving external sensors, the same object or motion is often expressed in different coordinate systems. In this project, three such frames are used:

- the base frame, which is fixed relative to the robot's mounting point;
- the tool (TCP) frame, which moves with the end effector;
- the camera frame, which is attached to the tool but displaced by a known offset.

The challenge arises because the robot accepts velocity commands in the base frame, but the input data is provided in the camera and tool frames. Without a consistent frame of reference, calculations of movement errors would be invalid, leading to incorrect or unstable motions.

Position-to-Velocity Conversion
The target-following task naturally defines a positional error: the difference between where the target is currently located and where it should appear relative to the tool. However, the robot controller does not accept positions as input—it requires velocities (SpeedL commands). This introduces the problem of mapping positional error into feasible velocity commands while ensuring that the motion is smooth and correctly aligned with the robot's kinematics.

Safety and Smoothness of Motion
Simply commanding the robot to move in proportion to the raw positional error can create unstable behavior. At large distances, this might produce unsafe velocities that exceed the robot's physical capabilities. Near the target, this can cause overshooting and oscillatory "wiggling," where the tool constantly moves back and forth around the goal. In addition to making the system inefficient, such motions can reduce the lifespan of mechanical components and compromise operational safety.

Real-Time Constraints
The system operates in a closed loop where new measurements arrive at each control cycle of duration dt. This means the computation of transformations, error vectors, and velocity commands must complete reliably within that time window. Any delay in execution would break synchronization between sensing and actuation, reducing tracking accuracy and potentially leading to unstable behavior.

More broadly, the requirement of smooth, continuous motion ruled out many of the built-in movement functions of the UR5e like predefined waypoints or “move to pose” commands. Instead, we had to implement a custom velocity-based control strategy from scratch, which required iterative testing and refinement.

As with many research projects, not all prior work was directly applicable.

Connection issues
Significant difficulties arose in manipulator control. Translating tracking errors into joint velocities required careful handling of coordinate transformations and velocity computations. Errors in matrix composition or numerical instability sometimes led to unstable motion of the arm. Furthermore, reliable communication with the robot presented an unexpected obstacle: our initial wireless setup caused intermittent delays and disconnections in command execution, making smooth velocity control nearly impossible. After extensive troubleshooting, we found in an old GitLab thread that others faced similar issues, and the solution was to switch to a wired Ethernet connection. This significantly improved stability and responsiveness.

Solutions - Detection System

For the detection stage, as stated before, we chose to rely on classical computer vision methods rather than deep learning approaches, since they were faster and more consistent.

1. Masking by Color (HSV vs. RGB)
The first step in our detection proccess is masking each frame based on predefined color ranges and gradients. We used the HSV color space instead of RGB because it separates color information (hue) from brightness (value). This made color detection more robust under varying lighting conditions, since changes in brightness affected the RGB channels strongly but had less impact on hue. As a result, our object detection based on color was more stable and reliable.

2. Contour Detection with OpenCV
After masking the frame, we detect object outlines using the cv2.findContours algorithm from OpenCV. This function extracts the boundaries of shapes in a binary image by choosing a white pixel and trailing the outline of the object it is a part of, allowing us to identify and track objects based on their visible contours. It was particularly useful for segmenting targets when color or motion cues alone were insufficient.

This approach makes the detection pipeline highly tailored to specific object types, since it filters first by color and then by shape.

Proof of Concept
As a proof of concept, we created a preset for tennis balls: the frame is masked according to their characteristic color, and circular contours are then detected within the masked region. Additional presets can be defined in the same way for other object categories, allowing the method to be extended to different use cases. This method provided a fast and stable tracking solution, making it well-suited for our purpose.

3. Tracking the Object
To overcome the challenges stated in the tracking stage, such as understanding the objects motion, we used the DeepSORT tracker, which combines Kalman Filters with a CNN-based appearance model. The Kalman Filter predicts the object’s next location, while the CNN helps select the most likely bounding box, by examining its contents and determining wether the object in the bounding box is the one being tracked, thus ensuring consistent and accurate tracking over time. Doing this allowes the tracker to keep a "track id" for each tracked object, and in this way we can track the wanted object.

Tracking Modes
Additionally, there are two distinct modes to selecting the tracked object, manual and automatic.
If the **manual** mode is chosen, the bounding boxes of all objects identified in the cameras view are shown, and any one of the objects can be chosen as the tracked object by pressing on its bounding box.
In **automatic** mode, the robot automatically chooses the object closest to the center of the frame as the tracked object.

After choosing the tracked object, whether using the manual or the automatic method, the robot enters tracking mode, in which the aformentioned detecting and tracking techniques are used on each frame in order to track the chosen object.

Losses and Edge cases
If the object goes out of the frame, or its tracking is lost for a moment due to high speed movement, the tracker sets the tracked object position to be the desired position. By doing this, once the object is lost the robot doesn't move in a straight direction with no object in sight to direct it, instead staying in its position.
Additionally, the tracked object's location is set to the desired location when the user is choosing the tracking mode, and choosing the object to track, thus the robot doesn't move during these stages.

Solutions - Control System

Frame Unification with Homogeneous Transformations
To resolve the coordinate system mismatch, the algorithm uses homogeneous transformation matrices. The target position is first expressed in the tool frame, then mapped into the base frame using the current tool pose. Homogeneous coordinates allow translation and rotation to be combined into a single matrix multiplication, ensuring computational efficiency and accuracy. By unifying all data into the base frame, the control law always operates consistently.

Error-to-Velocity Mapping
The system converts the positional error into a velocity command by dividing the displacement vector by the control timestep dt. This produces a velocity that, if followed for one cycle, would reduce the error directly. This approach effectively implements a proportional controller in velocity space, providing immediate responsiveness without the need for a complex control law.

Clamping and Slowdown Mechanisms
To enforce safety and smoothness, the algorithm introduces two safeguards:

- **Velocity clamping** ensures that the commanded velocity never exceeds the robot's maximum allowable linear speed. If the computed velocity is too large, it is scaled down proportionally while preserving direction.
- **Distance-based slowdown** scales the velocity according to the target's proximity. As the tool approaches the desired position, the allowed maximum velocity decreases, ensuring a gradual stop. Once the distance falls below a small threshold, the velocity is set to zero, preventing oscillations.

These mechanisms guarantee that the robot moves quickly when far from the target, but carefully and smoothly when near it.

Our controller instead of using PID, uses the following Custom Slowing Mechanism:

```
dist = np.linalg.norm(delta_pos)
max_lin = min(self.max_lin, self.max_lin * (dist / self.close_distance))
if dist < 0.01:
    max_lin = 0
```

It's parameters are:
close_distance: What distance is considered close enough for slowing down.
proportional function: Describes how the velocity should drop in relation to the distance from the target. We used - $dist \ / \ self.close\_distance$, but other can be used to achieve different behaviours.

PID vs. Custom Slowing Mechanism

The slowdown mechanism in our controller acts as a lightweight alternative to PID. Instead of combining proportional, integral, and derivative terms, it adjusts the allowed velocity directly based on the current distance from the goal. When the arm is far, it moves at full speed; as it approaches, the maximum velocity scales down smoothly with distance; and within a small tolerance, it stops entirely to avoid jitter. This behavior resembles a proportional controller but with a nonlinear gain schedule: the effective gain depends on the error size, shrinking near the goal rather than remaining constant as in PID. This ensures that the robot converges stably and smoothly without the risk of overshoot or oscillation.

An additional advantage of this approach is its flexibility. The scaling function does not need to be the simple ratio $dist \,/\, close\_distance$; any function of the error can be chosen to achieve different behaviors. For example, a quadratic or exponential decay could produce an even gentler slowdown near the goal, while a step-like function could enforce aggressive motion until very close to the target. This design choice allows us to tailor the convergence dynamics to the task, striking a balance between speed, safety, and smoothness. Compared to PID, which relies on fixed linear gains and requires careful tuning of three coupled parameters, this nonlinear mechanism provides a simpler yet more versatile way to shape the robot's motion at a high level

Efficient Real-Time Implementation

The transformations and velocity computations are implemented using lightweight matrix operations in NumPy. Homogeneous transformations are calculated through direct matrix multiplications, which are both mathematically rigorous and computationally efficient. This ensures that the entire pipeline-from sensor input to final velocity command-executes within the control timestep dt, maintaining real-time responsiveness.

Defense Mechanisms

To avoid collisions with the environment bounds, we relied on the robotic arm's built-in safety mechanism. This feature was generally effective, stopping movements when the arm reached its physical limits and preventing serious mishaps during operation. For most scenarios, this provided a reasonable safety net, ensuring that the arm did not damage itself or the surroundings.

However, in certain edge cases the built-in mechanism was not sufficient. For example, the arm could still attempt invalid movements before the safety stop engaged, which sometimes led to jerky motions or unstable behavior. Adding extra software-level defense mechanisms, such as real-time boundary checks, could prevent these issues by rejecting movement commands outside the permitted area before execution. Such proactive measures would not only enhance safety but also ensure smoother and more predictable operation of the system.

Detection Algorithm

Masking by Color

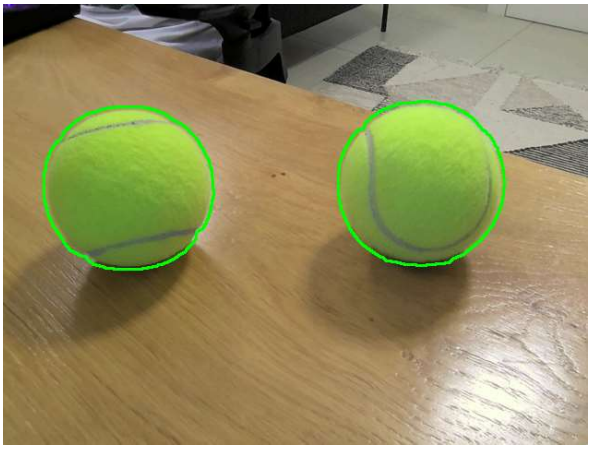
By masking the frame based on predefined color ranges and gradients, we can detect the shaped of the wanted objects in the frame. For example for our PoC for tennis balls, the following frame is masked:

And the results is:

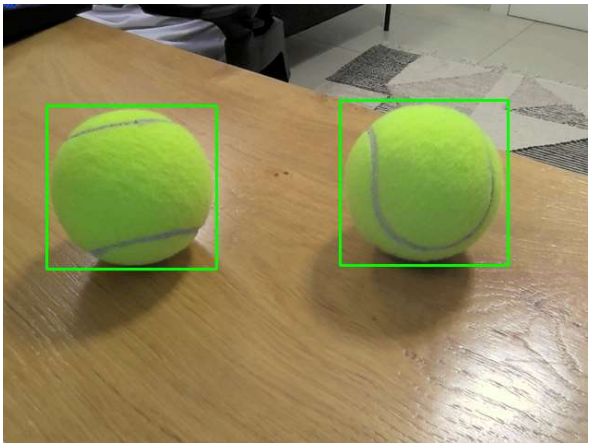
Contour Detection

We can see the shapes of the balls in white in the masked frame, and on this result we calculate the contours of the objects using cv2.findContours.

The reasult of that calculation is:

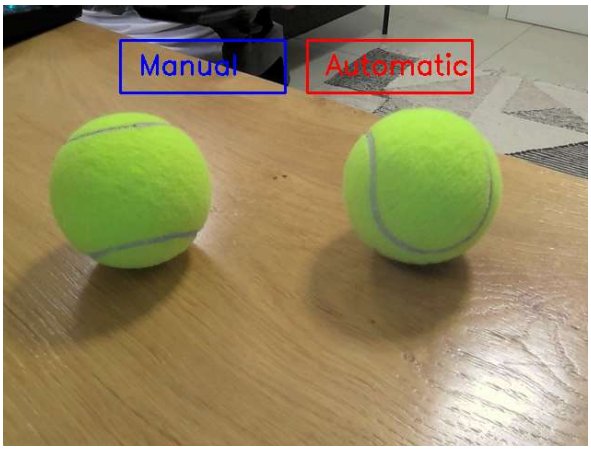


We then calulate the smallest bounding box around each conture:

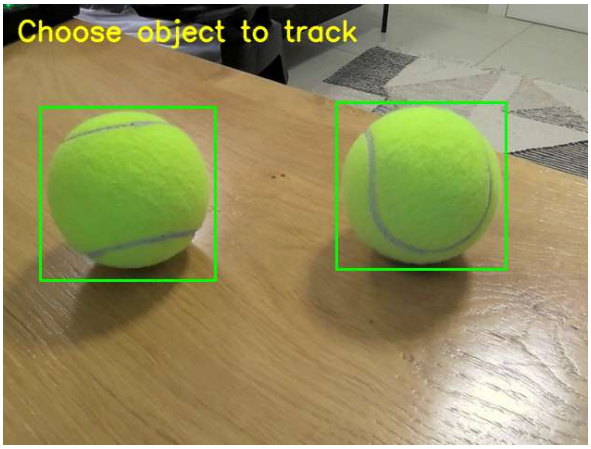


Tracking Modes

When loading the system the user is asked to choose between manual and automatic modes:

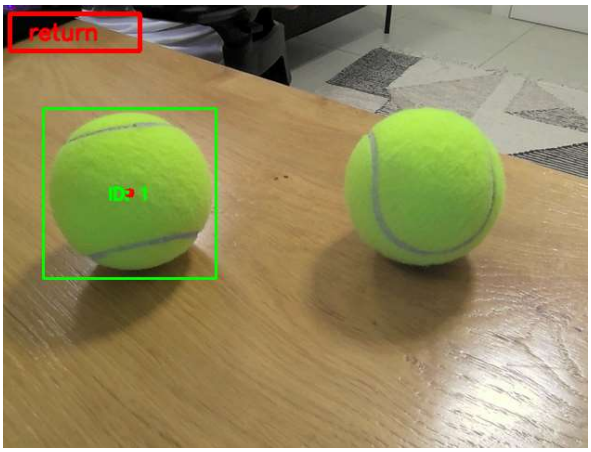


If the user chooses manual mode, he is requested to choose the object to track:



Otherwise, if the user chooses automatic mode, the object closest to the middle of the frame is chosen.

After choosing the object, or after selecting automatic mode, the chosen object's center is marked and its id is shown.

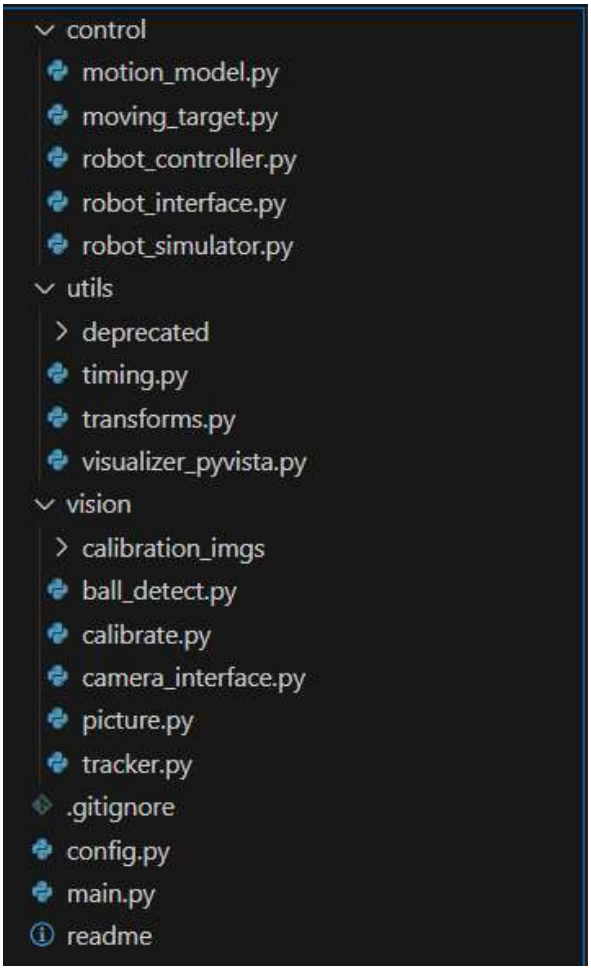


There also exists a return button, to allow the user to return to the beginning and choose a new tracking mode.

Results

A video demonstration the movment can be found in the github page of the project: https://github.com/GeckoWarrior/RoboticArm/blob/main/DEMO.mp4

Files Layout



Configuration:

The **config.py** file contains all configurable data, and seperated as follows:

SystemConfig:
mode - "live" for controlling the robot, "simulation" to control a simulated hand without robot connection.
target - "live" for using live feed for target, "simulation" for a simulated mocing target without vision.

RobotConfig:
Robot's ip, refresh rate, **desired position of the tracked object**, robot's speed limits and the offset of the mounted camera form the tool's center.

VisionConfig:
choosing camera (on robot or on laptop) and camera's intrinsic matrix.

Vision:

calibration_imgs: The images used to calibrate the camera, and obtain the K matrix.

ball_detect.py: The classical algorithm used to obtain the bounding boxes detections from the frame, this is a preset for tennis balls.

calibrate.py: The code used to calibrate the camera, using the images from Calibration_imgs.

camera_interface.py: The Camera class, responsible for extracting frames from the camera and providing them in a structured and convenient format for the tracker.

tracker.py: The Tracker class, responsible for identifying objects in each frame and tracking the chosen object. All the logic of the modes for choosing the tracked object are also in this class

Control:

motion_model.py: Defines the behavior of the robot's motion towards the target. We implemented one (Positional), but other types can be implemented as well (s.a. a rotational one, that keeps the end effector in the same position and only changes its facing towards the target)

moving_taget.py: Simulated dummy target to follow - for testing the control system without vision.

robot_controller.py: Higher-level logic, communicates the motion model's output to the robot via the intreface.

robot_interface.py: Wrapper for the RTDE control/receive.

robot_simulator.py: Simulates a robotic arm with a the same interface (for debugging / testing).

https://github.com/GeckoWarrior/RoboticArm.git

Retrospective

Advanced Deep Learning Models vs. Classical Algorithems
Looking back at our project, one of the main lessons was the trade-off between advanced methods and practical reliability. While deep learning approaches seemed promising, they required more time, training data, and hardware resources than were available. Simpler classical methods, while less powerful in theory, worked more consistently under our conditions. This highlighted the importance of matching the method to the task, rather than always choosing the most advanced option.

For detecting object outlines, we used the cv2.findContours algorithm from OpenCV. This function extracts the boundaries of shapes in a binary image, allowing us to identify and track objects based on their visible contours. It was particularly useful for segmenting targets when color or motion cues alone were insufficient.

Improving the Lab Setup and Knowledge Base
One of the biggest technical issues in the lab was the Wi-Fi connection between the control computer and the robotic arm. The network was not fast or stable enough to keep up with the high frequency of packets being sent. This often led to delays, dropped commands, or inconsistent responses from the arm. Using a faster and more reliable communication method, such as a direct wired Ethernet connection, would have provided much smoother operation and eliminated many of these interruptions.

Another difficulty came from the lack of a proper knowledge base. Instead of having clear documentation and guidelines, we had to rely heavily on trial and error. Most of the progress was achieved by searching online resources and piecing together solutions from various sources. This approach worked, but it was inefficient and left a lot of room for mistakes. A structured "information bank" would have saved time and made the workflow much more reliable.

Finally, the absence of centralized learning material also made onboarding new users harder. Without a proper reference, every student had to repeat the same trial-and-error process, which slowed down the overall progress of the project. A shared collection of tutorials, troubleshooting notes, and examples would not only speed up future work but also help ensure that solutions were consistent and reproducible across different projects and teams.

Python and UR-RTDE

The ur-rtde library is required for communication with the UR robot. However, setting up the library might be harder then a pip install... This is due to the fact the this library is originally written for c, and is wraped fpr python :)

Here are the steps for a setup that will (probably) work for you:

1. Set up a new envioronment for python (use conda) with python version 3.8.
2. In the env, run "pip install ur-rtde" - link: https://pypi.org/project/ur-rtde/

*** If the installation was completed successfully, you are one of the lucky ones.
If not, continue here:

3. Install "Boost": https://www.boost.org/
4. restart your computer.
5. Try "pip install ur-rtde" again.
6. If it still doesnt work, you might need to add cmake to the system path.
7. Try "pip install ur-rtde" again.
8. You are all set!

Future Work

Here we offer some ideas following this project:

Deep Learning for Object Detection

Initially we used YOLOv10 as the primary deep learning model for object detection. However, the results were not satisfactory, as the detector struggled with the diversity of objects in our environment, particularly since they were not limited to humans and included a broader set of items. This limitation reduced its reliability for our real-time application, so we reverted to classical computer vision methods, which, although less general, proved to be more consistent under our conditions. It is important to note that with more specialized training datasets, optimized architectures, and hardware acceleration, modern models such as YOLOv10 could potentially achieve robust detection across a wider variety of environments. The core challenge remains balancing detection accuracy and robustness with the strict real-time constraints of a closed-loop robotic control system.

Advanced Predictive Visual Tracking

A promising research direction lies in the development of advanced predictive visual tracking methods that go beyond simple motion extrapolation. Rather than relying solely on frame-to-frame displacement, such approaches would integrate probabilistic state estimation with dynamic motion models to forecast the target's trajectory under uncertainty. Additionally, incorporating deep recurrent architectures such as LSTMs or transformer-based temporal models would allow the system to capture long-term motion dependencies and adapt predictions to complex, nonlinear trajectories. This would enable the drone to anticipate not only linear movements but also abrupt changes in velocity, acceleration, or direction. In practice, advanced predictive tracking could reduce latency in control response, improve robustness against temporary occlusions, and provide a mathematically grounded framework for integrating perception and control in real time.

Multi-Modal Perception and Sensor Fusion

While our system relied primarily on visual feedback, future work could explore integrating multi-modal sensing for more robust perception. Combining RGB video with complementary inputs such as depth sensors and LiDAR could significantly improve the reliability of target detection and tracking, especially under challenging conditions like low light, glare, or visual occlusions. Sensor fusion algorithms could provide richer scene understanding by jointly reasoning over different data sources, thereby reducing reliance on a single modality. In a real-time robotic control context, this would not only enhance object localization accuracy but also provide redundancy, ensuring continued operation in the event of partial sensor failure or degraded visibility. The main challenge lies in developing lightweight fusion strategies that preserve the low-latency requirements of closed-loop control while exploiting the complementary strengths of multiple sensors.

References

- Wang, A., Chen, H., Liu, L., Chen, K., Lin, Z., Han, J., & Ding, G. (2024). YOLOv10: Real-time end-to-end object detection . arXiv.

- Wojke, N., Bewley, A., & Paulus, D. (2017). Simple online and realtime tracking with a deep association metric. 2017 IEEE International Conference on Image Processing (ICIP) , 3645–3649.