Robotic End Effector Tracking Challenge

Emily Sturman, [redacted]

December 15, 2023

1 Methodology

We implemented our code as a `computer_vision` module in `reacher` using OpenCV.

1.1 Camera Calibration

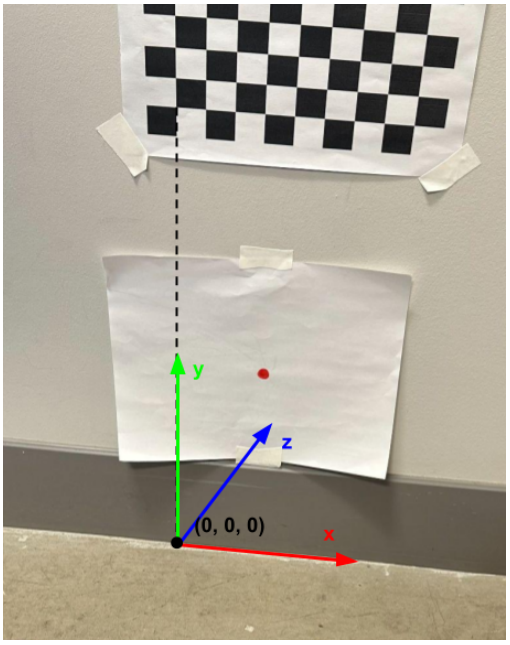

Figure 1: *World coordinate system*

The first step, before the main control loop, was to calibrate the camera. We used a printed chessboard taped to the same wall as the paper with the dot, and we assigned each chessboard corner a real-world coordinate (see Fig. 1). We used OpenCV's `findChessboardCorners`

to calibrate the camera from this matrix and compute the camera's intrinsic and extrinsic properties.

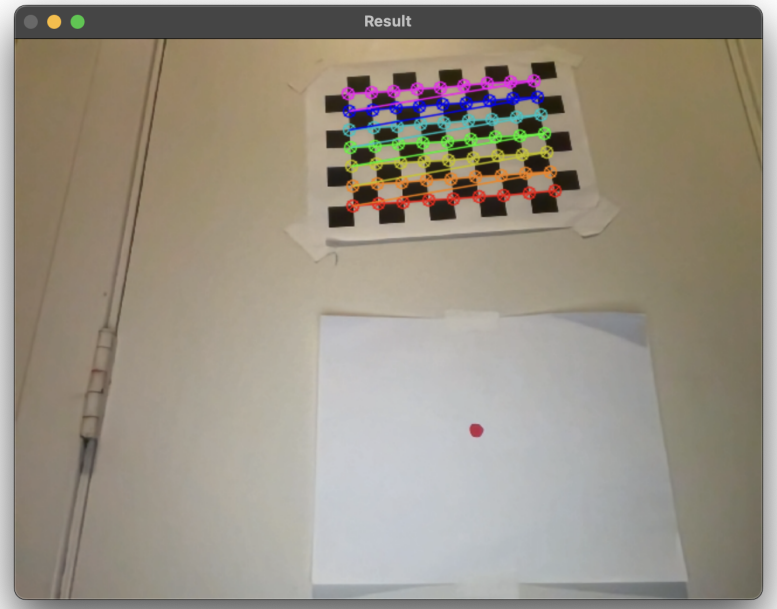

Figure 2: *Chessboard calibration*

1.2 Image Filtering

We located the red dot in the image using three criteria: color (hue), position, and size. We first masked the camera image by hue (filtering for slightly magenta to red), then dilated the image slightly so that the unmasked portions of the image can be grouped together into contours. We then filtered out all contours with a height or width greater than 20 pixels. The contour closest to the center of the image (by Euclidean distance) was then selected as the red dot.

1.3 Calculating World Coordinates

Through the image filtering process described above, we were able to arrive at pixel coordinates for the chosen red dot. In order to transform it to world coordinates, we started with

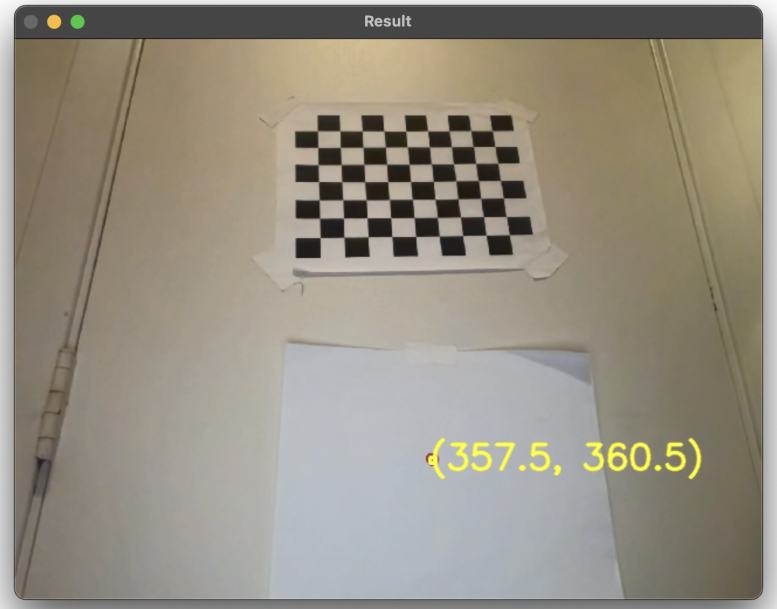

Figure 3: *Selected dot*

the following equation:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A \begin{bmatrix} R \mid t \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{1}$$

We already know (u, v) (the pixel coordinates), Z (0, the plane of the paper/chessboard), as well as our camera matrix A, rotation matrix R, and translation vector t (from camera calibration). We can manipulate equation (1) into the following:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A(R \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t)$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R^{-1}(sA^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} - t)$$

To compute s (our scaling factor), we can use the fact that Z is known:

$$\left(R^{-1}A^{-1}s\begin{bmatrix}u\\v\\1\end{bmatrix}\right)_{2,0} = \left(\begin{bmatrix}X\\Y\\Z\end{bmatrix} + R^{-1}t\right)_{2,0}$$

$$\left(R^{-1}A^{-1}\begin{bmatrix}u\\v\\1\end{bmatrix}\right)_{2,0} s = Z + (R^{-1}t)_{2,0}$$

$$s = \frac{Z + (R^{-1}t)_{2,0}}{\left(R^{-1}A^{-1}\begin{bmatrix}u\\v\\1\end{bmatrix}\right)_{2,0}}$$

Then, we have all the unknowns we need to compute $\begin{bmatrix}X\\Y\\Z\end{bmatrix}$.

1.4 Calculating Robot Coordinates

Once we computed the world coordinates, we simply need to convert them to robot coordinates. Given the difficulty of measuring angles in the physical world, we decided to always set up the robot with an angle of 0 (robot y-axis is directly perpendicular to the wall). Something else to note is that our computations were performed in centimeters and with the y-axis as vertical, whereas the robot receives inverse kinematics commands in meters and with the z-axis as vertical. Using all of this together, the robot coordinates can be computed as follows:

$$XYZ_{\text{robot}} = XZY_{\text{world}} - XZY_{\text{robot\_in\_world\_frame}}$$

1.5 Moving the Robot

Once we have computed the red dot in robot coordinates, we move the robot using our previously written inverse kinematics code. In order to filter outliers (detailed more in Challenges Faced), we keep track of the last 10 dot coordinates computed (approximately 0.1 seconds) and compute inverse kinematics from the median x, y, and z values.

2 Challenges Faced

2.1 Outliers and Color Similarities

One of the primary challenges we faced was regarding outlier dots being selected. Namely, the lower leg on our pupper arm was printed in magenta, which was often included in the color mask. Additionally, sometimes the groupings of these pixels would be small enough to bypass the size filters as well. While the position filter allowed the red dot to usually be selected, if the red dot was not detected in a frame for whatever reason, then a point on the leg could be mistakenly selected instead.

We had three approaches to fixing this. One (detailed above) was to select the median point among the recently computed dot coordinates, which allowed the robot to ignore these outliers. The second was to cover the robot arm in black paper, which allowed it to be excluded by the color filter in most cases. We also increased the dot size, which eliminated many small groups of red pixels that were spotted within the image (including the robot arm).

2.2 Range of Motion

The range of motion of the pupper arm was an extremely limiting factor while testing. We had to place the arm very close to the wall so that it would be able to reach the dot, and this often caused it to run into the wall while trying to reach the correct joint angle configuration. We experimented width different starting positions and configurations of the robot in relation to the dot in order to mitigate this, but it was ultimately an unavoidable constraint.

2.3 Covering the Red Dot

Another key issue we faced was when the red dot was covered by the pupper arm. While we did our best to prevent any red/magenta from being visible on the lower leg segment, our method was far from perfect, and there was often a spot or two that could still be seen. This led to the pupper being able to move to the red dot initially, but immediately moving away or back to starting position after misidentifying a spot on the arm as the red dot.

2.4 Return to Starting Position

The biggest challenge that we faced was that the robot returned to its starting position for any invalid inverse kinematics computation. This meant that if the dot was misidentified, the dot was covered, etc., the robot would immediately return to its home position (rather than following the dot). To address this, we disabled the feature and had the joint angles simply remain constant if the inverse kinematics returned incorrectly.

However, while testing this, we found that the robot would still move at extremely high speeds (which we later diagnosed as an issue with resetting to slider values). We attempted to limit the magnitude of the change in joint angles to approximately half of a revolution per second. However, upon testing this, our robot moved enough that it snapped both the lower leg segment and the hip joint segment.

Ultimately, we were able to fix this issue (using the TA's robot). We kept the inverse kinematics safeguards (i.e. resetting the joint angles for invalid IK results), but added another filter for outliers. After much trial and error, we realized that the red dot would occasionally be selected as a random point for one frame, too fast for the human eye to see. This was what caused the robot to attempt to return to home position so often. To fix this, we added another outlier safeguard: if the median value (calculated previously) was more than 5 centimeters away from the current `xyz` value, then we would not update the joint angles to match. This enabled the robot to smoothly follow the dot in simulation, and, once we replaced the inverse kinematics safeguard, we presume that it will also work on the physical robot.

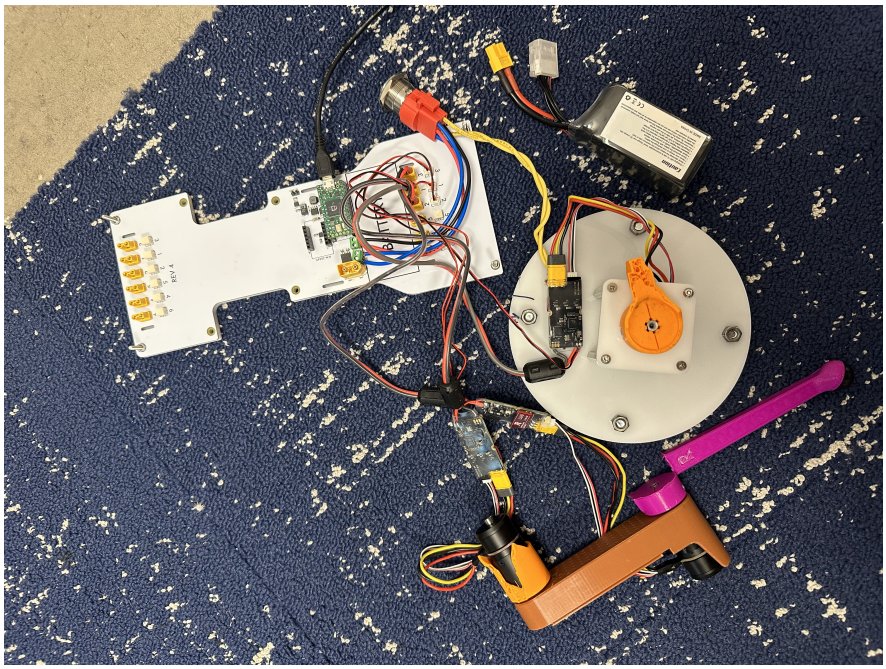

Figure 4: *Broken pupper robot*

3 Results

Given that our robot snapped, we were unable to record our final results on the physical robot. However, we have included a video of the code working in simulation, and we believe

that it will also work on a physical robot (if you are able to test on one).

If you are interested in testing on your own, the parameters that you will need to specify before running are:

- `ROBOT_X`, `ROBOT_Y`, `ROBOT_Z`: This is the distance from the world origin (see Fig. 1). Note that `ROBOT_Z` is the distance between the robot center and the wall (should be negative), while `ROBOT_Y` is the distance between the robot center and the floor.
- `CHECKER_SIZE`, `CHECKERBOARD_HEIGHT`, `CHECKER_WIDTH`, `CHECKER_HEIGHT`: `CHECKER_WIDTH` and `CHECKER_HEIGHT` should be the number of checkers in the board **minus one** (they represent the number of inner corners). `CHECKERBOARD_HEIGHT` should be distance between the floor and the inner bottom left corner (see Fig. 1).

Another important note is that the chessboard must be extremely close to the top of the frame (we would usually start with the top of the board off-frame, then slightly tilt the camera upwards until the board is registered). If this is not the case, then the camera may be calibrated upside down. To verify that this step is done correctly, the red dots should be at the top of the board, and magenta should be at the bottom.

You can view our code here, and our video here.