RoVi1

Final Project

Group: Petr Batěk, Bjarki Páll Sigurdsson, Salman Taj

December 15, 2016

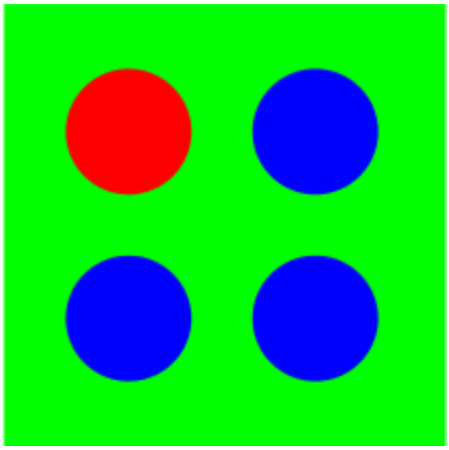


Figure 1: Marker 1.

Project Description

1 Feature Extraction

1.1 Marker 1

For this marker we decided to use color segmentation as our approach. The points we extracted were the centers of the three blue circles.

We began by converting the image to HSV colour space, splitting its channels and looking at the hue channel as seen in figure 2. We then took the hue value for the blue areas in the original marker as a reference. We used this value to compute a binary image showing only areas with a hue value very close to the desired blue one. The resulting binary image is shown in figure 3. This isolated the the blue circles from the marker very nicely but left some noise from the blue background elements. To ignore this noise, we took advantage of the circles' shape. We found the contours in this binary image and the radius of their minimum enclosing circle. By comparing this radius to the contour's area, we got a value which quantitatively described the roundness of the contour. By selecting the three roundest contours in the binary image, we obtained the three circles in the marker as seen in figure 4.

We used this marker to test the simulated visual servoing system. In order for the robot to follow the marker's orientation, the three extracted points must be sorted in a consistent manner. To implement this, we first made the assumption that the points form a triangle whose longest side is always the same one. While this is not true for all out-of-plane rotations, it holds for all the projections in the provided sequences. Following this assumption, we used the point opposite the long side as the first element and used the cross product of the two short sides to sort the remaining two points clockwise.

This algorithm finds the points with high precision on each image in the provided sequences. It handles in-plane and out-of-plane rotations and scaling but the colors are somewhat sensitive to ambient light changes. A blue ball or similar in the background

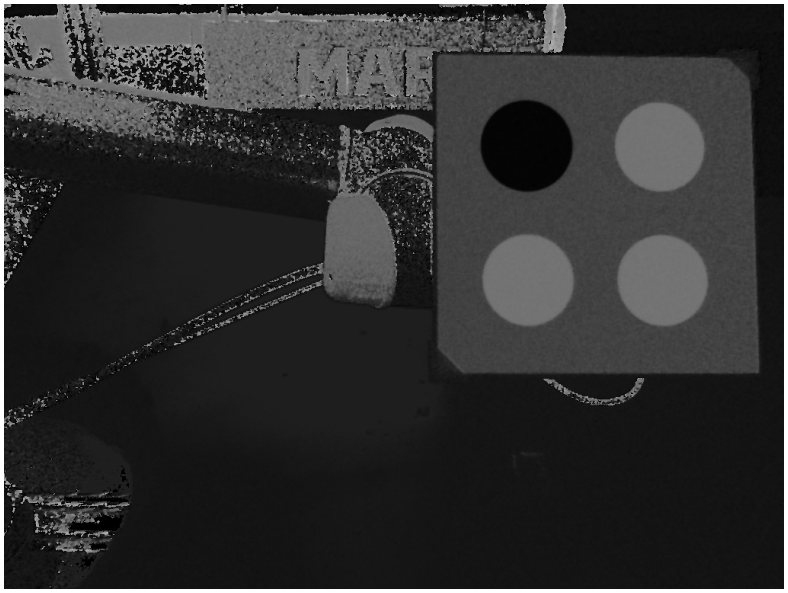


Figure 2: Hue channel of the first image in the color marker sequence.

would also be very troublesome.

The color segmentation approach is appropriate for this problem due to the sharp colors on the marker. Our method for circle detection is also quite robust for this problem due to the fact that the circles are well isolated and that we know beforehand how many to look for.

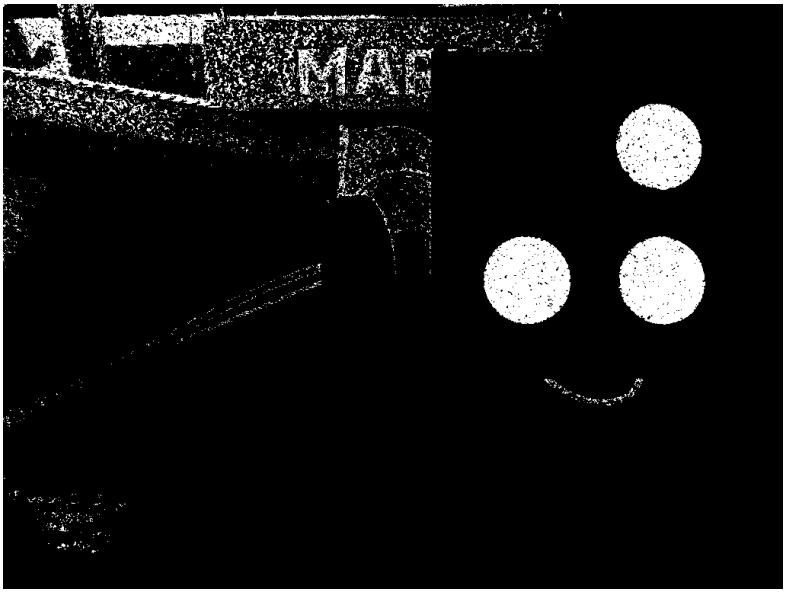


Figure 3: Figure 2 after hue thresholding.

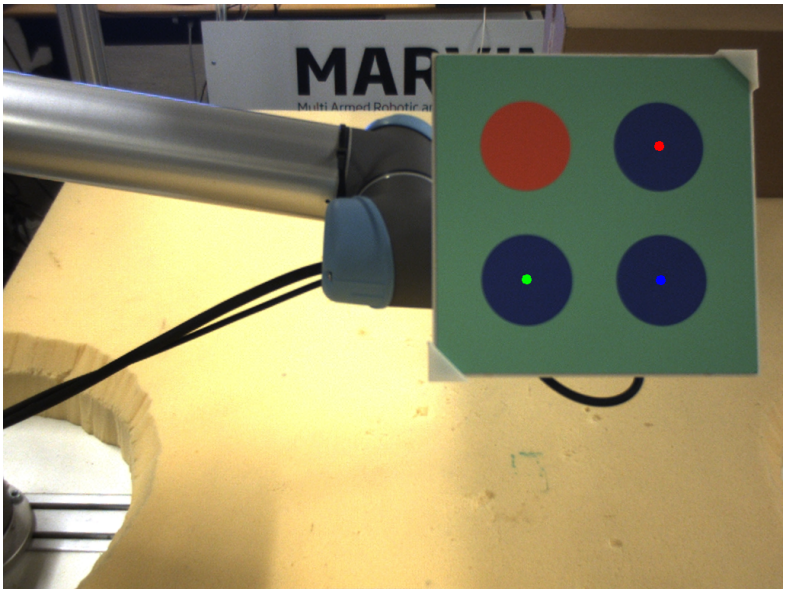


Figure 4: Extracted points.

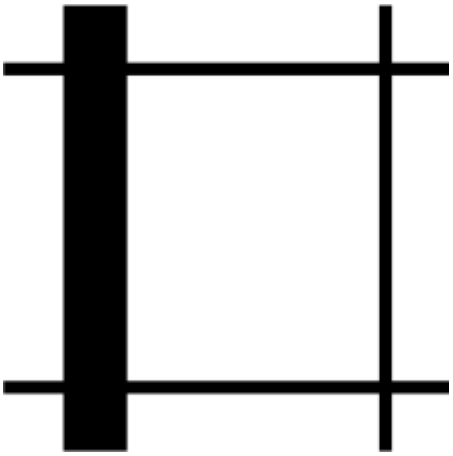


Figure 5: Marker 2b.

1.2 Marker 2b

For this marker we used the Hough transformation to find lines in the image. Our target points were the four corners of the white rectangle in the center of the marker.

We began by using the Canny algorithm to detect edges in the image as shown in figure 6. We also used the Hough transform to detect the lines as shown in figure 7. We dilated the edges as well as the lines and then combined them using logical AND as seen in figure 8. The resulting image shows the marker's lines relatively clearly with some background noise as well. From here we wanted to detect the innermost rectangle in the marker. We made the assumption that this rectangle is the largest one in this binary image. As for the first marker, we found the contours and this time their minimum enclosing rotated rectangle. This is quite a big approximation as it loses precision very quickly for out-of-plane rotations. To make sure we detect a rectangle, we compute a value which quantitatively describes how well the enclosing rectangle fits its corresponding contour. This eliminates most contours apart from the white rectangles in the marker. Finally, we pick the largest remaining rectangle which is our desired one. See figure 9.

This algorithm finds the desired rectangle for each image in the provided sequences but its precision is subpar for out-of-plane rotations as seen in [REFERENCE FIGURE]. The algorithm handles scaling and in-plane rotation but needs to find a closed contour around the desired rectangle. This makes it sensitive to sharp noise around the edges.

The Hough transformation is appropriate for this problem due to the long, sharp edges on the marker. We have chosen parameters for the line detection to maximize our chances to find the lines in the marker. We deal with the large number of false positives by ignoring the lines which don't correspond to edges in the image.

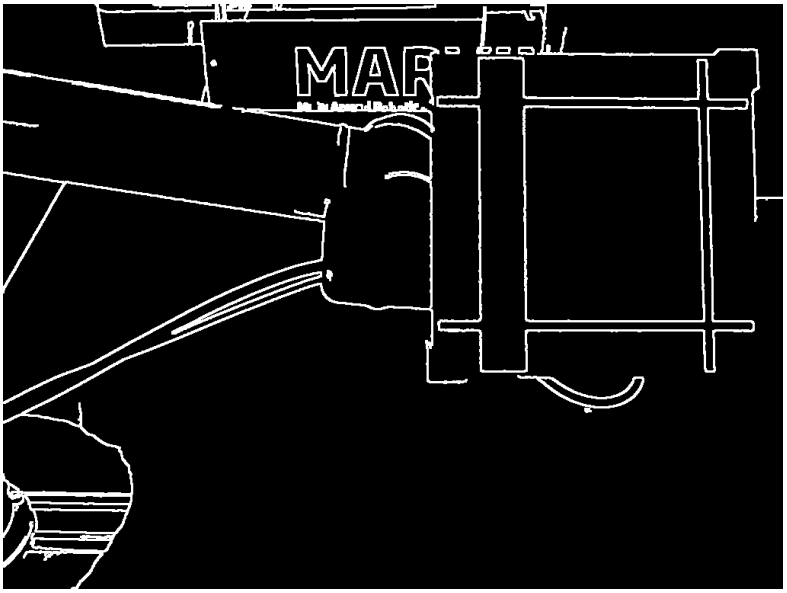


Figure 6: Edges from the first image in the thickline marker sequence.

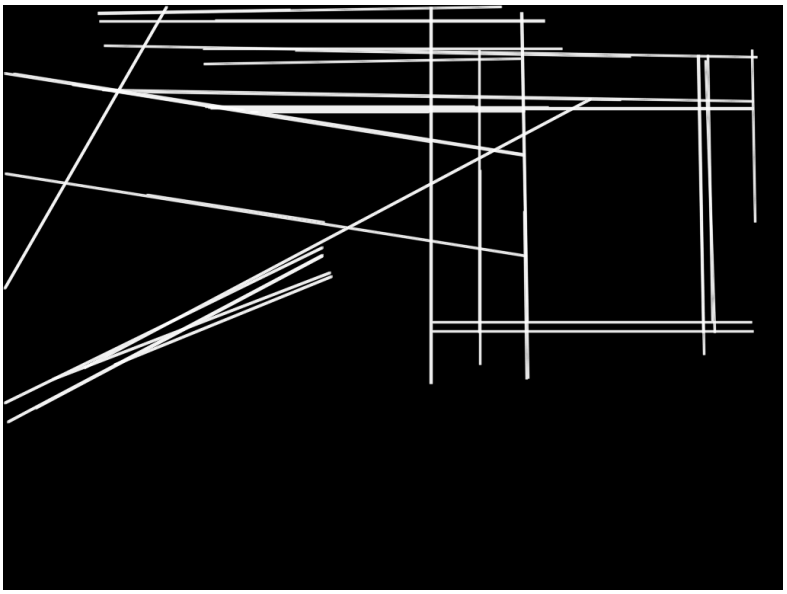


Figure 7: Hough lines from figure 6.

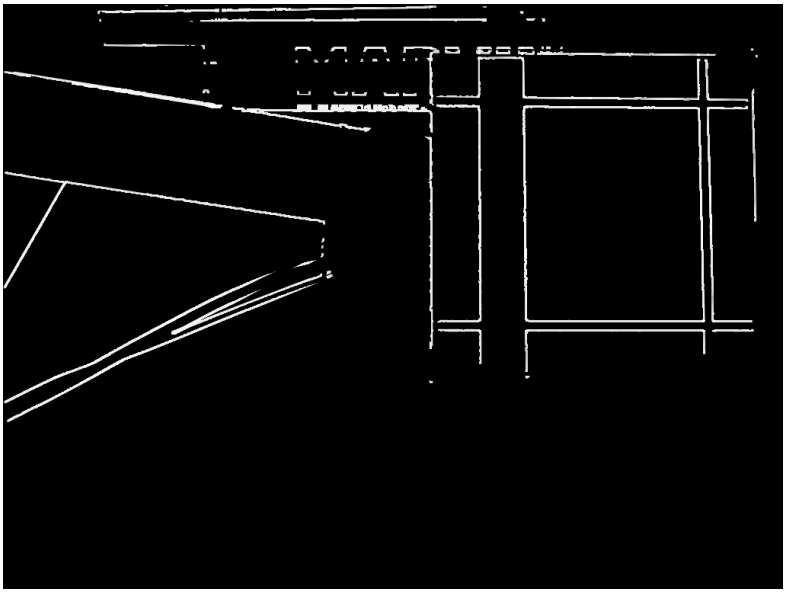


Figure 8: Logical AND of figures 6 and 7.

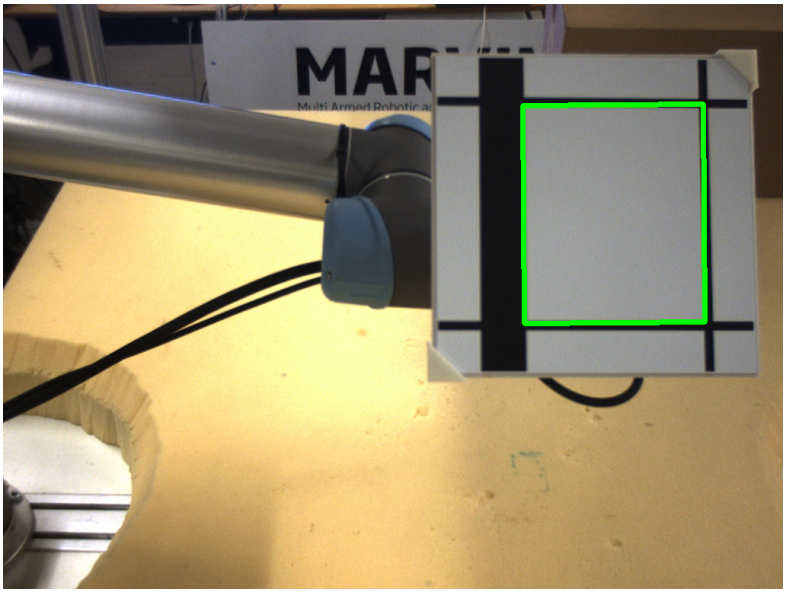


Figure 9: Rectangle consisting of the four extracted points.

2 Tracking points using image Jacobian

We have implemented algorithm for visual servoing in this part. For this part we selected specific marker points as described in problem statement and use mathematical camera model to get image pixel coordinates of the points. Image recognition thus wasn't used in this part.

To compute joint updates it was first needed to compose matrix $\boldsymbol{Z}_{image}$ as described in Robotics Notes:

$$\boldsymbol{Z}_{image}(\boldsymbol{q}) = \boldsymbol{J}_{image}\boldsymbol{S}(\boldsymbol{q})\boldsymbol{J}(\boldsymbol{q}) \tag{1}$$

where $\boldsymbol{J}(\boldsymbol{q})$ is manipulator Jacobian. For its computation we used function from RobWork library. $\boldsymbol{J}_{image}$ is image Jacobian matrix. We implemented function for its computation called `calculateImageJ` which can be found in the file `inverseKinematics.cpp`. We used fixed value for z coordinate. Since we used frame `cameraSim` to model the camera we set $z = -0.5$ for every function call. Finally matrix $\boldsymbol{S}(\boldsymbol{q})$ was composed by inserting transpose of the rotational matrix $\boldsymbol{R}_{base}^{tool}$ twice to its diagonal.

The next information necessary to compute joints updates is difference or move of target points $\overrightarrow{\boldsymbol{dU}}_{image}$. We have programmed function `calculate_dUImage` to solve for $\overrightarrow{\boldsymbol{dU}}_{image}$.

Having matrix $\boldsymbol{Z}_{image}$ and $\overrightarrow{\boldsymbol{dU}}_{image}$ it was possible to solve for joint positional update $\boldsymbol{dq}$ using method of Linear Least Squares.

We adapted two equations from robotics notes into single expression for $\boldsymbol{dq}$ computation:

$$\boldsymbol{dq} = \boldsymbol{Z}^T \left(\boldsymbol{Z}\boldsymbol{Z}^T\right)^{-1} \overrightarrow{\boldsymbol{dU}}_{image} \tag{2}$$

In this equation we used $\boldsymbol{Z}$ to denote $\boldsymbol{Z}_{image}$. For this solving of this LSM problem we have implemented function `compute_dQ_LSM` which can be found in `inverseKinematics.cpp`.

`algorithm2` is the function, where are all of described functions organized together to compute joint updates $\boldsymbol{dq}$ base on the manipulator state and the error in image coordinates $\overrightarrow{\boldsymbol{dU}}_{image}$.

We have implemented J_{image} and $\overrightarrow{\boldsymbol{dU}}_{image}$ composition in a scalable way, so the same functions can be used to track one or multiple target points.

Model of the robot manipulator has velocity limits on joint movements, so it was necessary to check if the limits were satisfied before joint updates. We did so by measuring time for inverse kinematics computations τ_1, subtracting this time from workcell update period specified by $\Delta T \longleftrightarrow$ `deltaT` and finally we divided update $\boldsymbol{dq}$ by the result of subtraction. Velocity of joint movement is the result of the operation:

$$\dot{\boldsymbol{q}} = \frac{\boldsymbol{dq}}{dt} = \frac{\boldsymbol{dq}}{\Delta T - \tau_1} \tag{3}$$

By comparing the actual velocity with manipulator limits it was possible to find out if the limits are satisfied. If they aren't algorithm simply saturate joint movement in

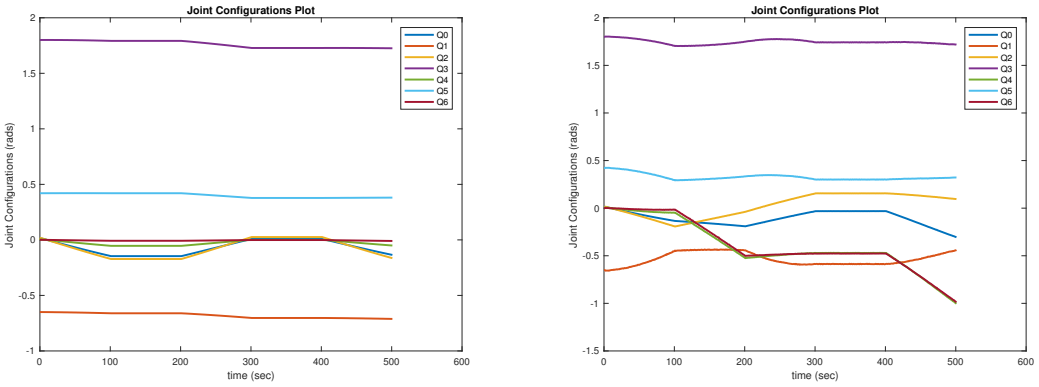
order to hold all conditions. For comparison and saturation, function `saturateDQ` was implemented.

In the following section we provide simulation results from tests of inverse kinematics.

2.1 Simulation Tests

During simulations, we recorded joint configurations, tool/camera frame position and orientation for `deltaT` $= 1000ms$ and finally we performed tests for different values for `deltaT` in the range $50\,ms <$ `deltaT` $< 1000\,ms$ and plotted maximum errors of $\overrightarrow{\boldsymbol{dU}}_{image}$

Slow Marker Sequence



(a) Tracking Single Point

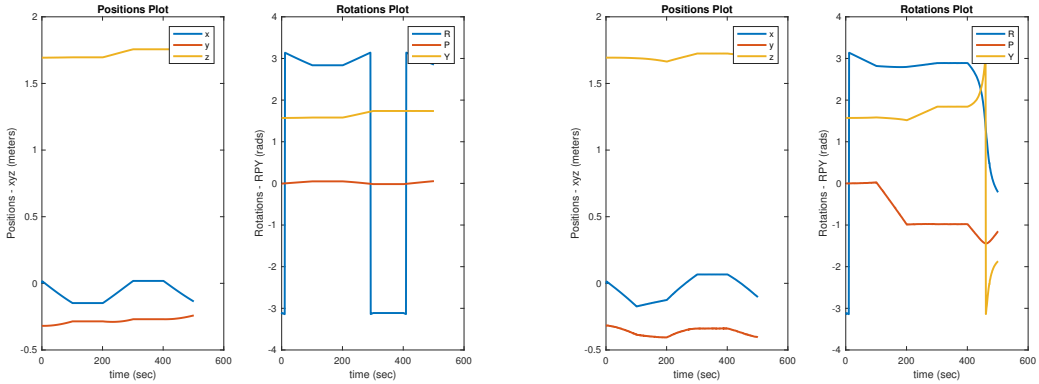
(b) Tracking 3 Points

Figure 10: Joint configurations

There are differences between joint coordinates for tracking single and 3 target points in the graphs 10a and 10b. The reason behind this is following. When the manipulator is tracking single point, it just follow its position. There is no orientation information about the marker when using single tracking point. Whereas during following of 3 target points, orientation of the marker gains important role, as the manipulator is trying to rotate its tool/camera frame to align its position and orientation with the marker.

In figure 11a and 11b are apparent jumps in roll angle around z axis. This however doesn't imply jumps in tool orientation. Abrupt jumps in the graphs were caused because of switching between $-\pi$ and π rad angle. In the real world, change in orientation is small. Implementation of Robwork transformations probably keeps all orientation angles in the interval $\langle -\pi,\ \pi \rangle$.

Simulation for different ΔTs



(a) Tracking Single Point

(b) Tracking 3 Points

Figure 11: Tool pose transformations

Simulation for different ΔTs

There are plotted maximum pixel errors on the figure 12.

For timing purposes we used C++ Chrono library, for taking time instant of start of the computation of updates and the end instant. For differentiation of joint updates and subsequently getting of velocity, we used difference of these two time stamps divided time which left for joints update.

As you can see on the figure 12 we didn't reach manipulator velocity limits in the interval $\Delta T \in \langle 0.05, 1 \rangle \, [s]$ as was described in the problem statement.

For this reason we lowered the interval to the $\Delta T \in \langle 0.005, 0.02 \rangle \, [s]$. And in this case, there is apparent increase in the maximum errors for low ΔT. Results are plotted in the figure 13.

Due to non-real time OS, results might be different for each simulation and doesn't offer precise information. However for the purpose of the school exercise, we were able to limit joints speed velocities according to problem statement.

Medium Marker Sequence

The same problems and explanation apply for the Medium marker sequence. Joint coordinates and tool positions are plotted on the graphs 14 and 15. Only difference from Slow sequence is in the time axis. Due to faster and larger movements time span is shorter.

Simulation for different ΔTs

There are plotted maximum pixel errors on the figure 16. Again, we didn't reach manipulator velocity limits in the original interval. Results after lowering the interval of ΔT are plotted in the figure 17.

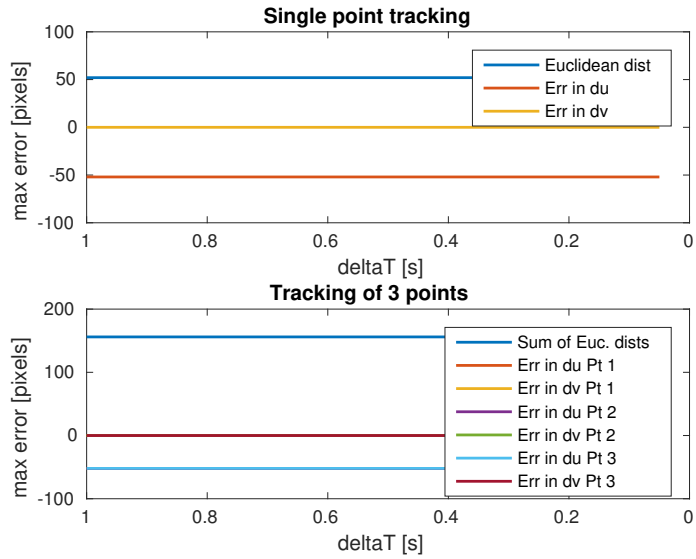


Figure 12: Maximum errors during Slow Sequence Marker following

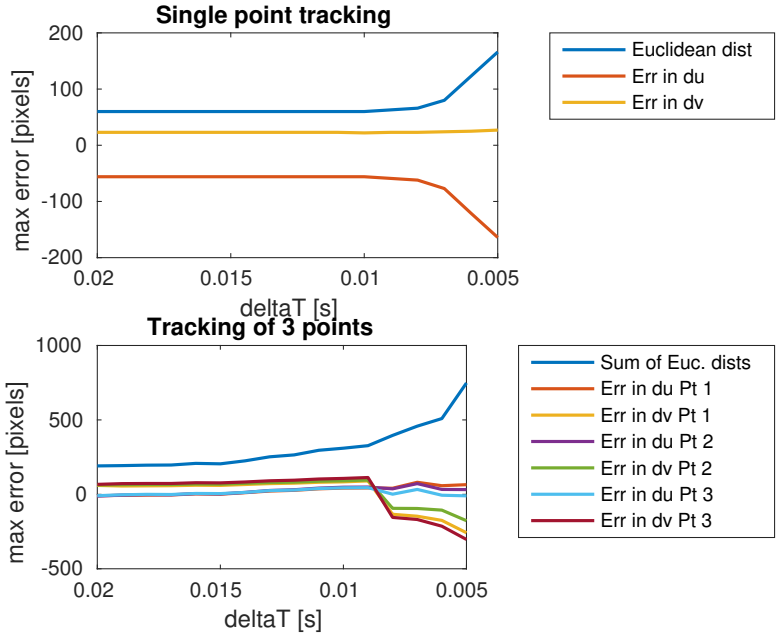
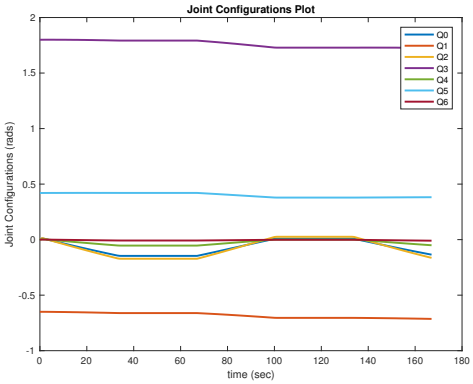
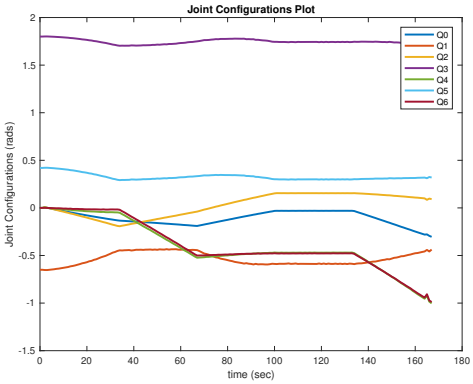


Figure 13: Maximum errors during Slow Sequence Marker following

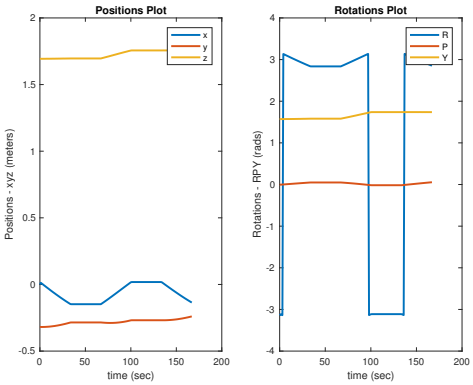


(a) Tracking Single Point

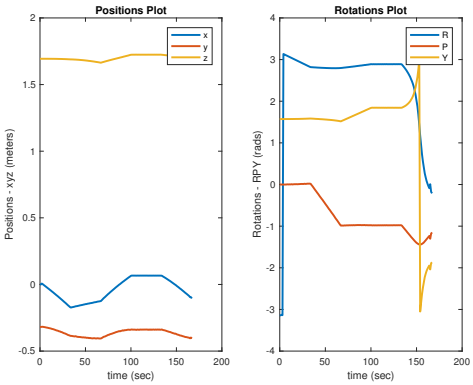


(b) Tracking 3 Points

Figure 14: Joint configurations



(a) Tracking Single Point



(b) Tracking 3 Points

Figure 15: Tool pose transformations

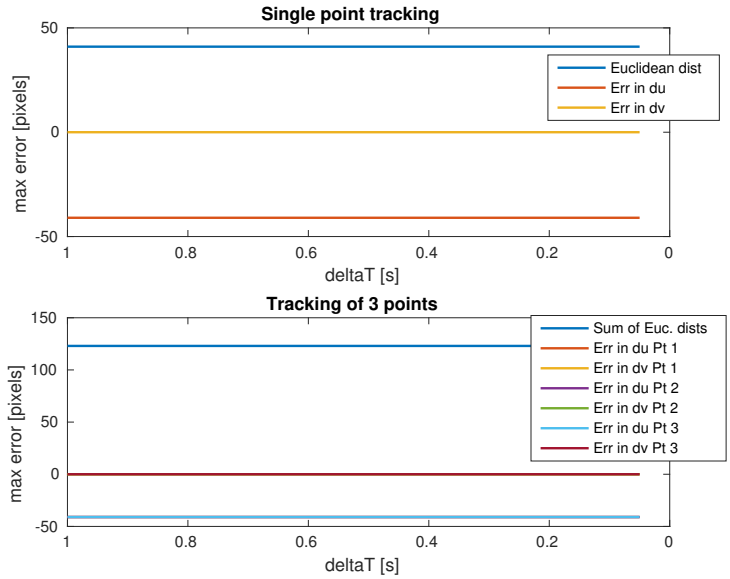


Figure 16: Maximum errors during Medium Sequence Marker following

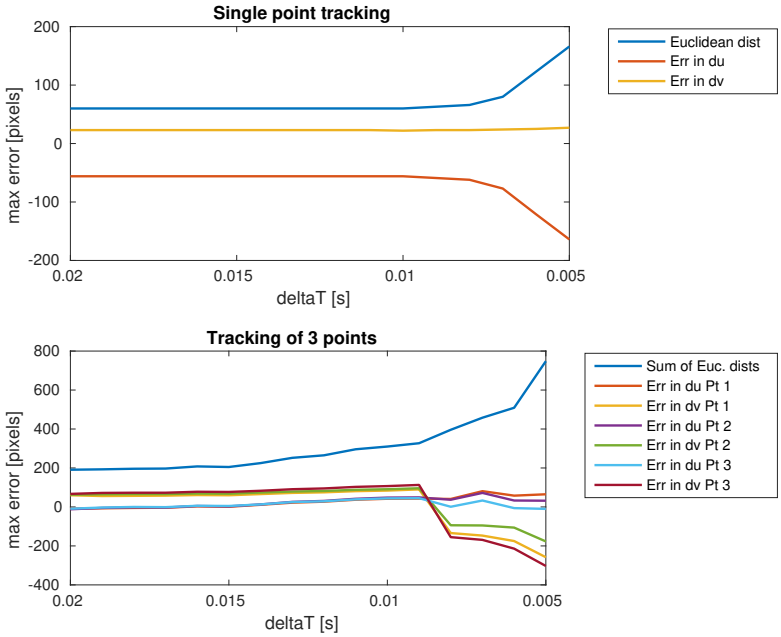
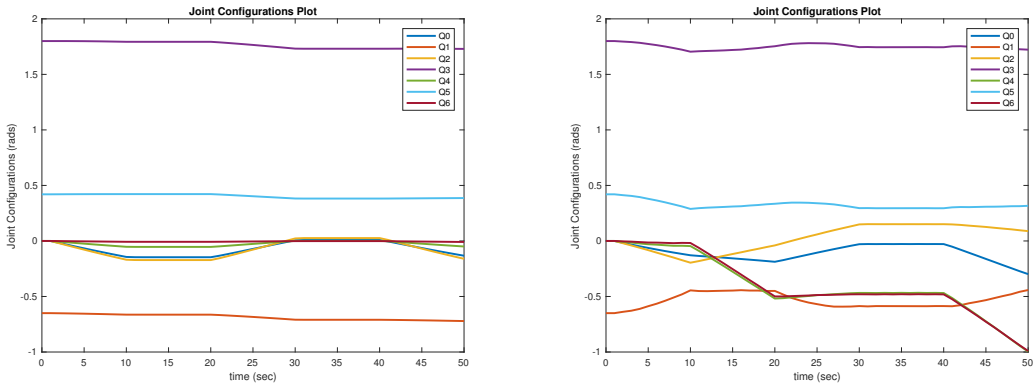


Figure 17: Maximum errors during Medium Sequence Marker following

We have to state the same as for the preceding test. All timing tests were performed on the regular laptop PC, running non-real time operating system. So results from another tests can differ.

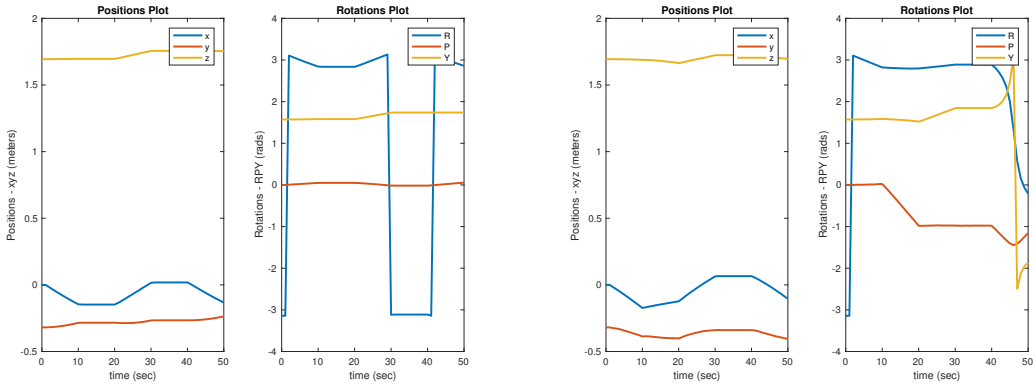
Fast Marker Sequence

Joint coordinates and tool positions are plotted on the graphs 18 and 19. Span of time axis is again shorter.



(a) Tracking Single Point (b) Tracking 3 Points

Figure 18: Joint configurations



(a) Tracking Single Point (b) Tracking 3 Points

Figure 19: Tool pose transformations

Simulation for different ΔTs

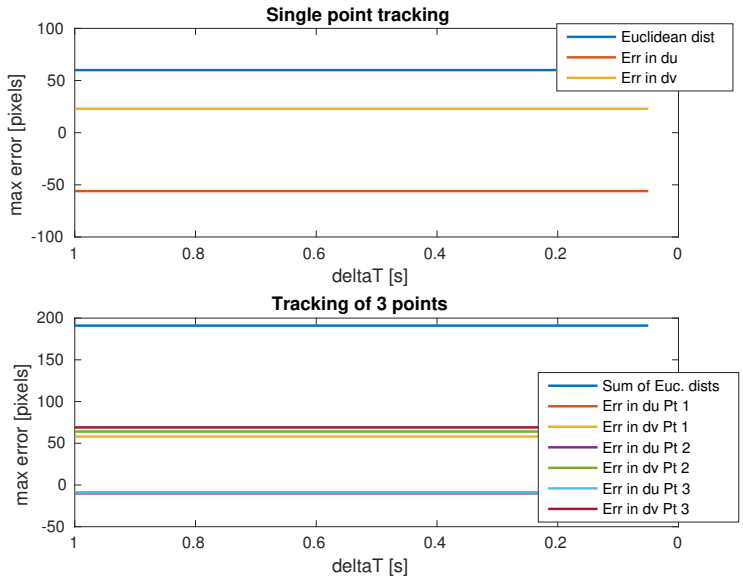


Figure 20: Maximum errors during Fast Sequence Marker following

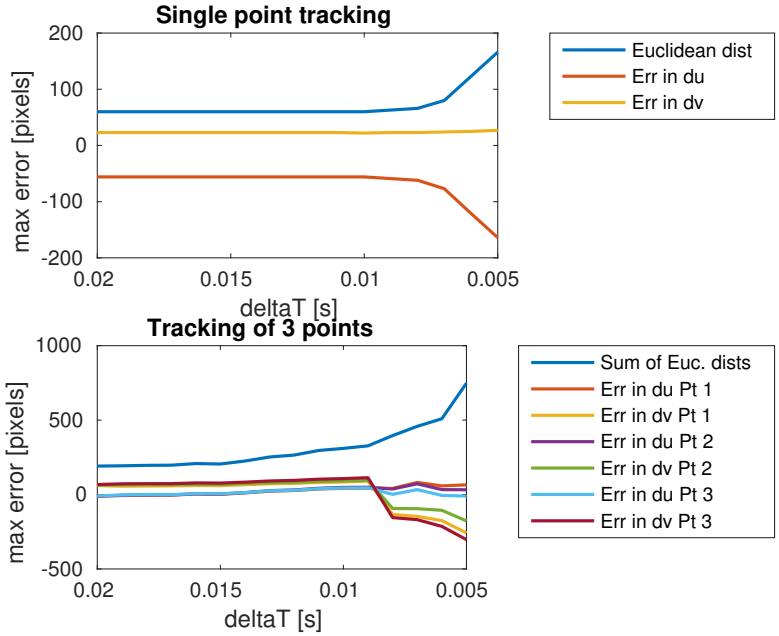


Figure 21: Maximum errors during Fast Sequence Marker following

There are plotted maximum pixel errors on the figure 20 for original interval. Results from simulations for lower interval are plotted on the figure 21.

Conclusion for different speed sequences

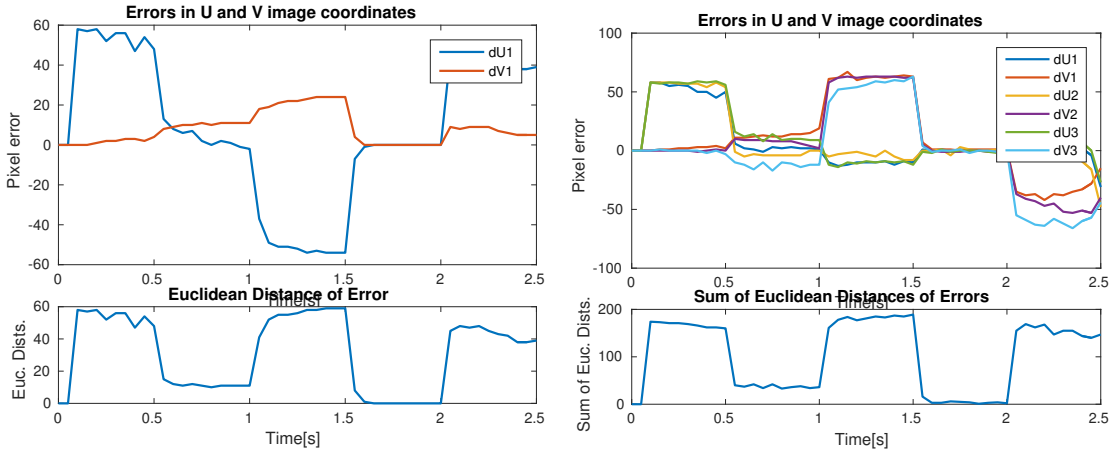
Even though timing and velocity computation during test simulations wasn't precise due to non-realtime of the OS, we can conclude some final facts. Manipulator was able to follow the marker for each sequence with `deltaT` $> 0.01s$. The reason behind these results is probably very short time needed for inverse kinematics computations. We found out is in the order of microseconds, whereas simulation were performed for ΔTs which were one order in magnitude larger (miliseconds). These results will be also different when simulated on different computer.

3 Combining feature extraction and tracking

For the last part of the project we have chosen the Marker 1 for image recognition. Image recognition was integrated into Robwork project. All the image recognition function are in the file `marker1.cpp`. For proper tracking of multiple points, it was necessary to sort the detected points in consistent order for every detection.

After performing of several tests we have found, that recognition time varies among different machines. Average time for one of our computer was around 30 ms whereas on the second one it was just around 25 ms.

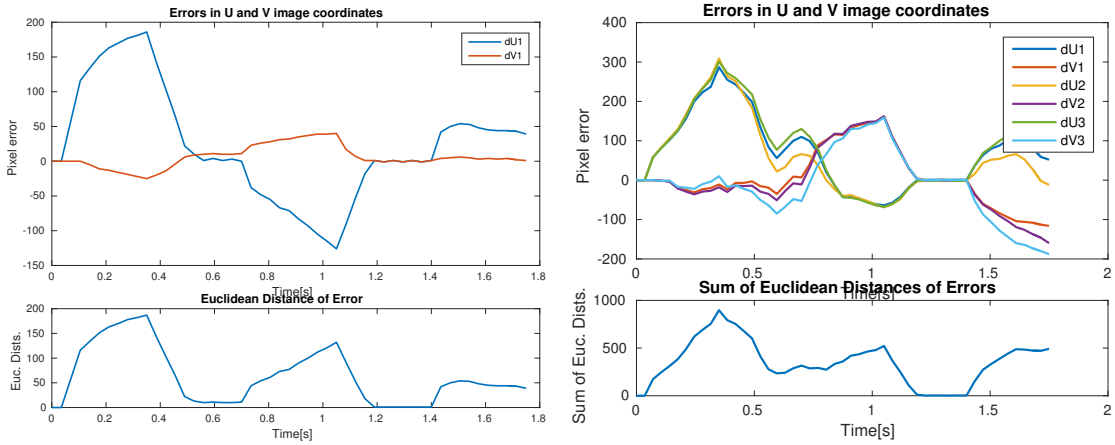
For further plots we have chosen one `deltaT` which don't cause any problems during following. And the second one, which is too short and unsaturated joints velocities are larger than limits.



(a) Tracking Single Point

(b) Tracking 3 Points

Figure 22: Slow Sequence, `deltaT = 50` ms

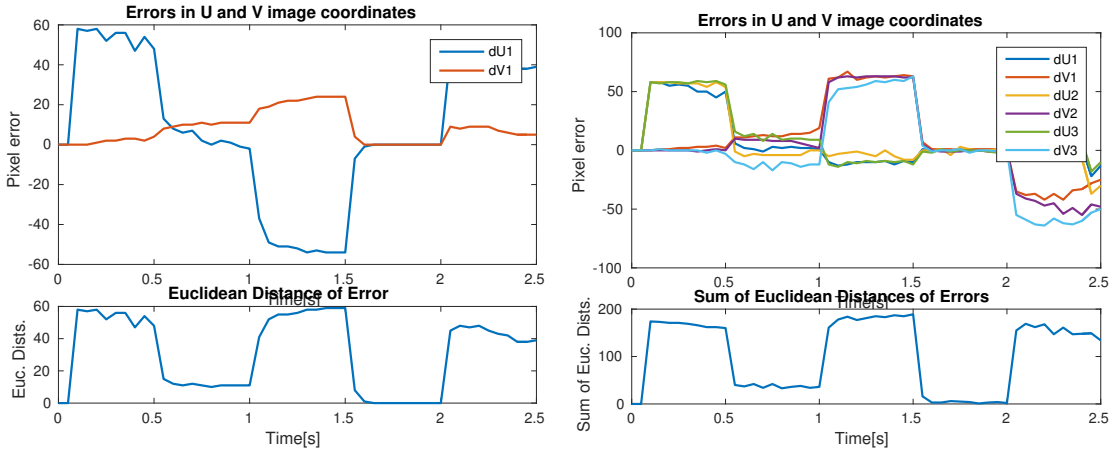


(a) Tracking Single Point (b) Tracking 3 Points

Figure 23: Slow Sequence, `deltaT = 35` ms

It is possible to see, that tracking error is generally larger on the figure 23 than on the 22. The reason is that the velocity limits were violated several times for `deltaT` = 35 ms. Another interesting find out about errors is the shape of the errors in the figure 22. The error signal shapes rectangular function. The error is larger during following the linear movement of the marker than during rotational times.

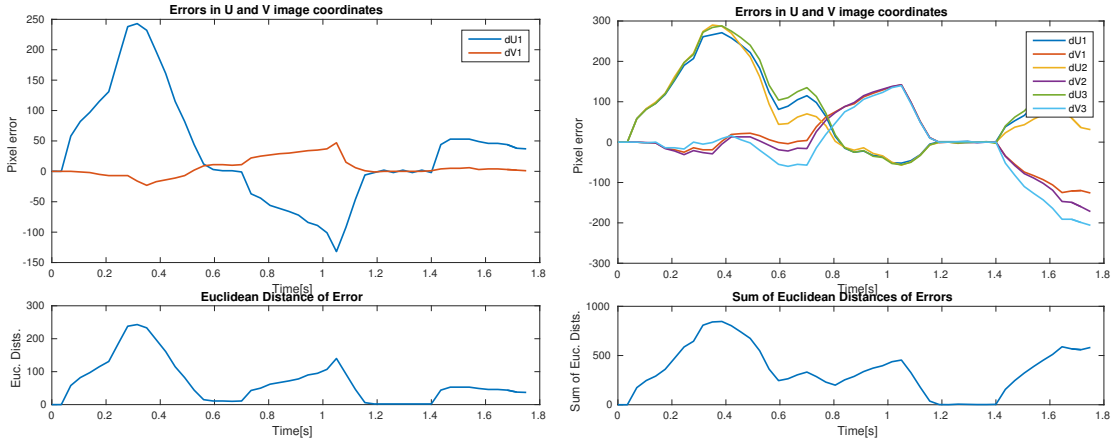
Similar reasoning applies also for marker Medium sequence on the figures 24 and 25.



(a) Tracking Single Point (b) Tracking 3 Points

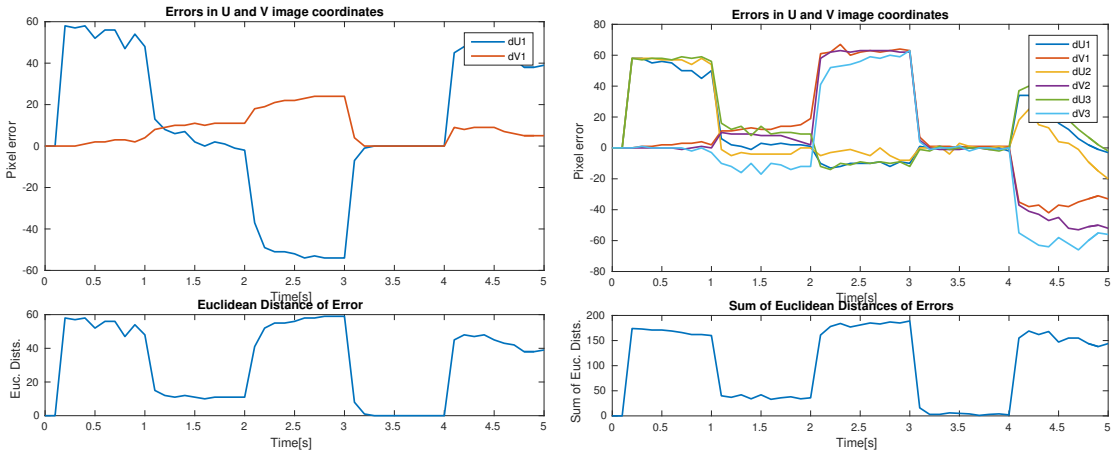
Figure 24: Medium Sequence, `deltaT = 50` ms

Finally simulation for Fast sequence are plotted on the 26 and 27.



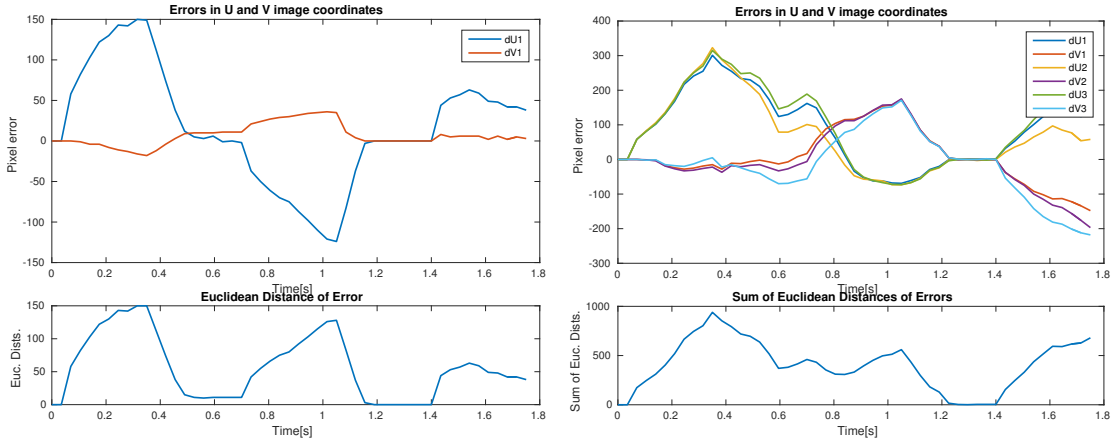
(a) Tracking Single Point (b) Tracking 3 Points

Figure 25: Medium Sequence, `deltaT = 35` ms



(a) Tracking Single Point (b) Tracking 3 Points

Figure 26: Fast Sequence, `deltaT = 100` ms



(a) Tracking Single Point (b) Tracking 3 Points

Figure 27: Fast Sequence, `deltaT = 35` ms

Scaled joint positions and velocities