Introduction to Robotics and Computer Vision
-
Final Project

Visual Servoing

Edited By

Anders Ellinge
aelli14@student.sdu.dk
Mathias Gregersen
magre14@student.sdu.dk

The University Southen Denmark
Odense

SDU

2017

1 Introduction

The purpose of this project was to apply knowledge and skills acquired from various lectures from the *Introduction to Robotics and Computer Vision* course, and develop a visual servoing application (eye-in-hand). That is, a camera mounted on a robot arm tracking a specified object. The project has been divided up into three main categories: **1.** Feature Extraction **2.** Tracking Points Using The Image Jacobian **3.** Combining Feature Extraction and Tracking

Feature Extraction	With the project three markers were provided. The goal was then to apply methods to find/identify two of these three markers in a picture provided by some camera. Furthermore test images for each of the markers was also provided in this project. I.e. a easy and a hard sequence of robot arm configuration holding a marker in a real life environment.
Tracking Points Using The Image Jacobian	Given one or more points in image space, the task was then to track this/these points using a robot control algorithm. Data sets with medium and fast robot motion sequences was also given with the project, upon which the robot control algorithm could be tested.
Combining Feature Extraction and Tracking	A sample plugin for RobworkStudio was provided with the project. In this plugin the Feature Extraction and Tracking parts were combined to create and test the Visual Servoing in a simulated environment in RobworkStudio.

2 Feature Extraction

Being able to identify desired features from a image is a strong tool. In this project the task was to identify at least three points on two different markers. On figure 1 the two markers chosen in this project, from a provided list of markers, are depicted. The three points of interest for marker 1, see figure 1a, was decided to be the center of the red circle and the two perpendicular neighboring blue circles. For marker 3, see figure 1b, the three points of interest was decided to be the bottom left, top left and top right corners of the marker. These choices of reference points for both markers will become more apparent when revealing the methods in section 2.1. After explaining the methods the test results of the methods will be shown in section 2.2 and finally in section 2.3 a discussion of the methods and test results will take place.

(a) Marker 1 (b) Marker 3

Figure 1: The markers which will be used for feature extraction.

2.1 Methods

This section will discuss the methods used to extract features from the chosen markers.

2.1.1 Marker 1

The method chosen to extract desired features from marker 1 was done by color segmentation of red and blue followed by creating contours of the found colors. The image moments of the contours were then calculated and from these moments it is possible to calculate the central moment, i.e. the center point of the contour which are the desired reference points. See figure 2 which shows the methods applied to detecting the blue circles of marker 1.
The color segmentation was done by HSV(Hue, Saturation, Value) segmentation with the function *segmentateHSV()* found in the file *ip.cpp* in the *plugin/src* directory. The function works by thresholding the picture to a binary image with a desired HSV range of colors converted to white and everything else black. Just using segmentation with HSV is not enough to get a "clean" segmentation of the desired color range. Artifacts from the environment will very likely contain parts of the desired HSV range which needs to be filtered. This is done with closing and opening the image, which is just a combination of dilution and erosion in a correct order, to remove unwanted "not big enough" blobs of the desired color range. At first glance it would seem odd to apply closing, because it fill gaps and strengthens the unwanted parts from the environment. The reason for doing this, is that the markers may contain spots of colors that failed to lie within the desired range, so those gaps needs to be filled. After the color segmentation and filtering, the contours of the resulting white areas are then calculated and followed by a calculation of the image moments of said contours. Creating the contours and image moments was done using the OpenCV library functions REFERENCE. The central moment of each contour is then computed with the *getCenterPoints()* function from the file *ip.cpp*, the function simply applies $\bar{x} = \frac{M_{10}}{M_{00}}$ and $\bar{y} = \frac{M_{01}}{M_{00}}$, where M_{ik} denotes a moment.

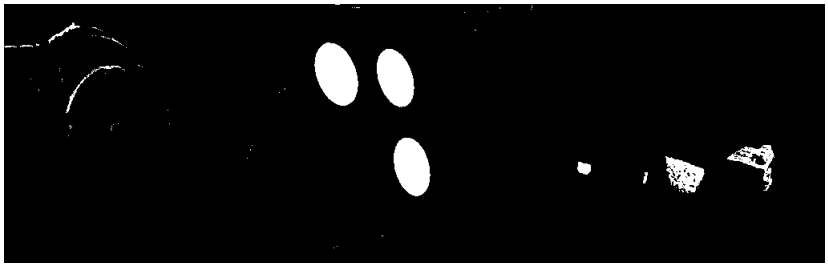
There will be more center points when segmenting the blue circles on the marker. The center points of these blue circles are not necessarily ordered with regards to the marker, since they are found when scanning the picture from the bottom right corner going up column-wise. E.g. when the marker is rotated enough, the center points of the blue circles has to be sorted with regard to a reference point. An obvious reference would be the red center point, since there should only be one red center point. To sort the blue center points with regard to the red center point the following method was implemented.
Create three vectors, $\overrightarrow{RB_1}$, $\overrightarrow{RB_2}$ and $\overrightarrow{RB_3}$, where R and B denotes the red and blue center points found from the marker, see figure 3. Then chose a vector pair which dot product is $min(\overrightarrow{RB_1} \cdot \overrightarrow{RB_2}, \overrightarrow{RB_1} \cdot \overrightarrow{RB_3}, \overrightarrow{RB_2} \cdot \overrightarrow{RB_3})$. This pair will be the two vectors which are closet to being orthogonal, i.e. the vector not in this pair should lie between the pair. Now calculate the cross product of this pair, let's say $\overrightarrow{RB_1} \times \overrightarrow{RB_2}$ like the examples in figure 3. If this product is positive then the order of blue center points should be (B_1, B_2, B_3), like on figure 3a, or if negative (B_2, B_1, B_3) like on figure 3b. An implementation of this method can be seen in function *decideOnBlueMarkers()* in the *ip.cpp* file in the *plugin/src* folder.

2.1.2 Marker 3

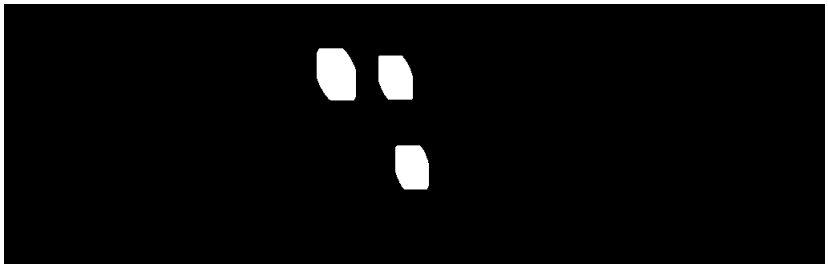
Marker 3 is a very unique marker without any obvious geometry or color to track. Due to this, a more advanced feature detection method had to be implemented. It should be noted that the implemented method is heavily inspired by the OpenCV feature2d module tutorials [3]. OpenCV provides functions and implementations for feature extraction method, which otherwise would have been out of scope for this project to implement, therefore significantly easing the process used. To track marker 3 a SURF(Speeded Up Robust Features) [6] detector was used to create keypoints and descriptions of them on the marker. These descriptions are then matched with descriptions of similar keypoints in the scene where the desired marker is present. Good matches are collected and used to create a homography between the matched keypoints. This homography is then used to make a perspective transformation of the marker's chosen reference points to the scene, thereby identifying the desired markers reference points in the scene. On figure 4 the reference points, using the method for marker 3, has been found from a scene in the hard sequence of pictures provided with the project.

All this is done by the *marker3Function()* in the MARKER 3 ????? ALGORITHM FILE in the WHATEVER ????? DIRECTORY. Before creating the SURF detector a minHessian value has to be decided. OpenCV uses this value upon construction of the SURF detector.The simple explanation of what the minHessian is used to, is that it thresholds the detection of interest points [6]. A minHessian of size 400 was chosen, as suggested by

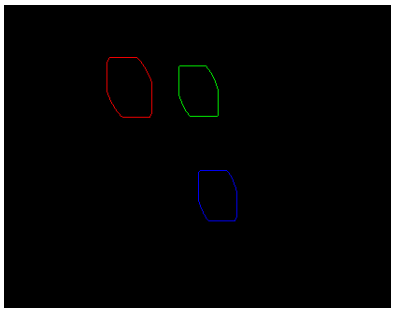
(a) Crop of one of the "harder" test images from the hard sequence



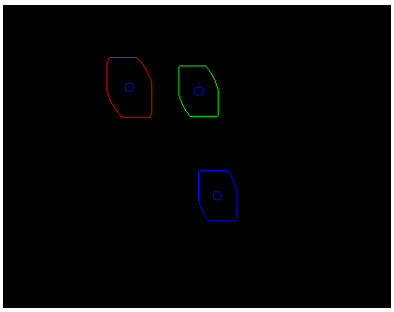
(b) Result of applying HSV color segmentation with regards to a desired blue range



(c) After applying closing and opening on the segmented image only areas of interest are left



(d) Found contours after closing and opening the segmented image



(e) The central moments found, i.e. desired reference points

Figure 2: Images showing the steps in the blue color segmentation method applied to identify reference points. It should be noted that the crops on the images were done after running the algorithm on the test images.

OpenCV [4], which seems to work satisfyingly. The SURF detector outperforms other discussed methods in the course, like SIFT, both in speed and accuracy [6]. SURF detects keypoints and also creates descriptors, which makes the implementation for feature extraction straight forward.Calculating the keypoints and descriptors is as simple as calling the *detectAndCompute()* function on the detector from the OpenCV library with a scene

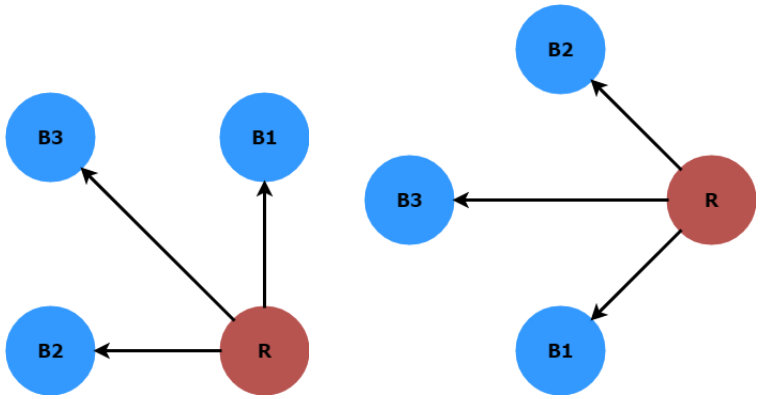


Figure 3: An example of finding blue center points of the contours in two different ways because of the order they are found in.

Figure 4: An example of finding points (circles in black) when applying the method for marker 3 to one of the pictures in the hard sequence provided.

image or object image. That is, a set of keypoints and descriptors need to be created for both the object image you wish to identify and the scene image containing the object, which should be detected.The matcher used is a Flann(Fast Library for Approximate Nearest Neighbors) based matcher. It creates Kd trees to index the descriptors and uses a nearest search method to find the best matches. The Flann based matcher may therefore be faster than a brute-force method when doing matching on large sets of descriptors [2].To only keep the good matches the maximum and minimum distances between matches are defined. Then keeping the matches with a distance of less than three times the minimum distance is defined as a good match [4]. Extracting the coordinate point from the good matches of keypoints of both the object and the scene and then passing them to the function *findHomography()* [1], which creates a homography between the points in the object and the points in the scene. Furthermore *findHomography()* also needs a method to fit a homography from one data set to the other. The method used to achieve this was RANSAC, which OpenCV supplies. If a homography is found, then that homography is used to create a perspective transform of the reference corner points in the object image. This is done with the *perspectiveTransform()* [5] function from the OpenCV library,

HSV ranges.

Color	Hue range	Saturation range	Value range
Red	[0, 10]	[137, 215]	[80, 215]
Blue	[105, 125]	[83, 180]	[42, 115]

Table 1: HSV ranges used to segment the blue and red colors from marker 1 on the provided easy/hard sequences of test images.

Test results easy sequence

Marker	Success rate	Max Error	Avg. Execution time	Std. deviation
Marker 1	100%	≈ 1	19,12	1,86
Marker 3	100%	≈ 14	325,83	46,61

Table 2: Test results of the two implemented methods applied to the easy sequence test images provided with the project.

that simply takes a set of points and uses a provided perspective transform to transform the points.Now it is simply a matter of using the points gotten by the method.

2.2 Test Results

The methods for marker 1 and marker 3 were tested on set of easy/hard sequences provided with the project for each marker. In DIRECTORY ????? WHATNOT and DIRECTORY ????? SNOT in the attached project files two scripts, *runMarkerTest* and *runMarkerTestHard*, will automatically run and generate test data on the provided sequences when run. The scripts was used to generate the test result data on a virtual machine running on a PC with a Intel(R) Core(TM) i5-6600K CPU @ 3.50GHz. In table 2 the results of the easy sequences of marker 1 and 3 are shown, likewise on table 3 are the results of the hard sequences. The success rate denotes how many of the pictures successfully found the marker or not, that is all the desired points were found and confirmed to be right. Max error denotes the approximate maximum error a found point deviated from the real point in pixels, this data was gathered by manually looking for errors on points printed on the scene and are therefore prone to human error. The average execution time of the methods is measured in milliseconds and likewise the standard deviation of the runtime is also measured in milliseconds.

When testing marker 1 on the sequences the HSV ranges used to segment the colors are given in table 1.

2.3 Discussion

2.3.1 Marker 1 Discussion

The method developed for marker 1 is very sensitive to environmental changes such as illumination and general "noise" in the background. This was exactly the case when trying the algorithm on the test sequences. Quite a lot of tinkering with HSV values had to be done to make the algorithm work on all the pictures in the different sequences, and when a larger HSV range is used the more "noise" will appear and reduce the final results precession.

Test results hard sequence

Marker	Succes rate	Max Error	Avg. Runtime	Std. deviation
Marker 1	100%	≈ 7	18,57	2,35
Marker 3	100%	≈ 14	399,48	26,94

Table 3: Test results of the two implemented methods applied to the hard sequence test images provided with the project.

The method used for marker 1 is not invariant to projective transformations. This is because that when features are extracted from the marker with a skewed perspective, the circles will appear as ellipses. The centers of these ellipses will be found with the central moments, but these are not the true centers of the circles. The true center of the circle would be shifted in either direction depending on the projective transform from circle to ellipsoid. This invariance to projective transforms will become apparent in section 4. When marker 1 changes perspective, the robot might opt to simply move closer to the marker instead of correcting to a perspective change.

In conclusion: The method implemented for marker 1 is a Very fast method, but not very robust. It is only applicable to very specific markers. It has problems with perspective transformation and is generally only recommended when execution has to be fast or to be used in a controlled environment.

2.3.2 Marker 3 Discussion

The method used to track the desired features in marker 3 is very robust. It should be robust to illumination, rotation, location and scaled changes [6]. While the method for marker 1 has problems with projective transform, the method for marker 4 should be invariant to projective transformations up to a Viewpoint angle from 40 to 60 degrees [6]. Though as seen in the tests, the method for marker 3 is significantly slower than the method for marker 1.

In conclusion: The method implemented for marker 3 is quite slow, but should be robust. The method is applicable in most situations and for other markers as well. If execution time is not required to be very fast, then the method for marker 3 is a good choice.

3 Tracking Points Using The Image Jacobian

This section goes through the implementation of the visual tracking part of the project. First we go through how we calculate the dq based on a single point and then extend it to multiple points. Lastly We test the functionality on the given motion sets.

3.1 Calculating dq for a single point

Using the theory from the 4.9 section in the robotics notes, we see that calculating the dq value involves calculating Z_{image}, displacement vector of the the displacement in u and v (du and dv), and using the Moore-Penrose inverse to get a least norm solution. Calculating the Z_{image} involves calculating the image Jacobian (J_{image}), the conversion of **du** to the base frame (S(q)) and the device Jacobian (J(q)). Then equation used for Z_{image} can be seen on equation 1. The J_{image} is calculated using equation 2, the S(q) is calculated using equation 3 and the J(q) is simply gotten from the device in RobWork. The du and dv were calculated by taking the difference between the point where the marker is at, and where we want it to be (The origin of the tracking). Using equations 4 and 5 we are able to solve for a dq using the Moore-Penrose inverse. We use the Moore-Penrose inverse functionality from the RobWork library given in the namespace LinearAlgebra. After getting a dq we also need to apply the velocity limits of the robot. This can be done by simply getting the velocity limits from the device and then comparing them. Here it is important that you transform the dq values to be in the same time space as the velocity limits (which is 1 second). This is done by dividing the dq with the time that the movement is gonna take (delta_T). It is important to note that if the velocity limit is broken the maximum velocity is chosen. This however needs to be converted to the time space that the original dq is from, which is done by multiplying with delta_T.

$$Z_{image}(q) = J_{image} S(q) J(q) \tag{1}$$

$$\begin{bmatrix} du \\ dv \end{bmatrix} = \begin{bmatrix} -\frac{f}{z} & 0 & \frac{u}{z} & \frac{uv}{f} & -\frac{f^2+u^2}{f} & v \\ 0 & -\frac{f}{z} & \frac{v}{z} & \frac{f^2+v^2}{f} & -\frac{uv}{f} & -u \end{bmatrix} = J_{image} \cdot \mathbf{du} \tag{2}$$

$$\mathbf{du} = \begin{bmatrix} [R^{cam}_{base}(q)]^T & | & 0 \\ -- & & -- \\ 0 & | & [R^{cam}_{base}(q)]^T \end{bmatrix} \mathbf{du}^{base} = S(q)\mathbf{du}^{base} \tag{3}$$

$$(Z_{image}(q)[Z_{image}(q)]^T])y = \begin{bmatrix} du \\ dv \end{bmatrix} \tag{4}$$

$$dq = [Z_{image}(q)]^T y \tag{5}$$

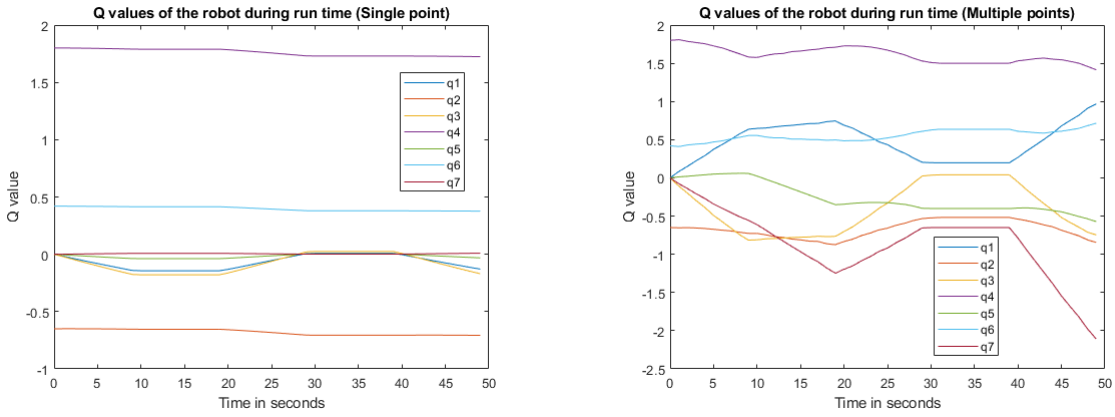
3.2 Calculating dq for multiple points

When calculating dq for multiple points the J_{image} and the displacement vector is the only affected part of the calculations. The way that J_{image} is calculated therefore needs to be updated to handle multiple points. This is done by simple stacking the equations. The same is done for the displacement vector. Be advised that if 2 times the number of points tracked is larger than number of joints in th device, the solution should be found using another technique then the Moore-Penrose inverse. In this project we tracked only 3 points, meaning that the Moore-Penrose inverse would suffice.

3.3 Testing the functionality

The tests have been performed on the given motion sets for the marker. Getting the U and V values for the tests was done using the pinhole model to calculate these from the marker position. In order to get multiple points for measuring the WorkCell file was extended with 3 new frames parented to the marker frame. These three new frames was given a transform with no change in orientation and a positional change related to the project description. The positional changes were (0.15, 0.15, 0), (-0.15, 0.15, 0) and (0.15, -0.15, 0). The U and V from these new frames where calculated using the pinhole model, just like with the singular point.

In figure 5a the changing robot configuration when tracking a single point can be seen when running on the fast marker motion and a delta_T of 1 second. The tool pose for the robot from the same test can be seen on figure 6. These figures arbitrarily shows the movement of the robot and the tool. This test was conducted with a delta_T of 1 second. The same test was run with three points and the resulting robot configurations and tool pose can be seen on figure 5b and 7.

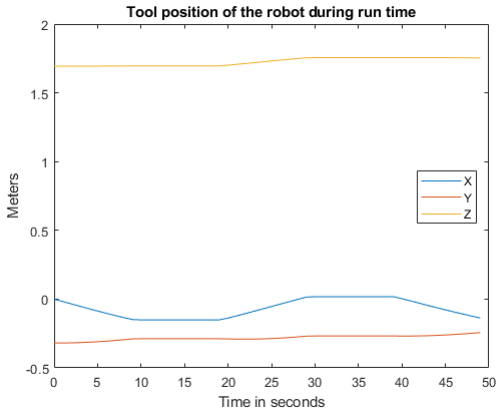


(a) Q values for a single point

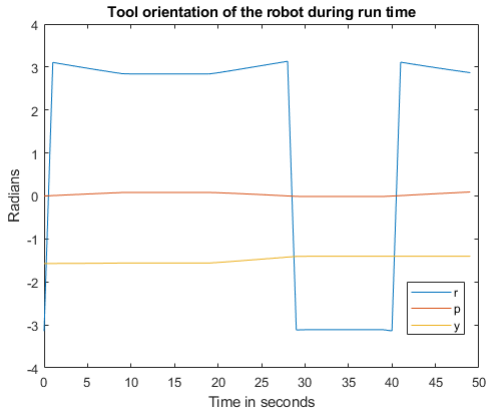
(b) Q values for multiple points

Figure 5: Q values of the robot during a run with a delta_T of 1 second on the fast marker motion.

Testing the effect of the delta_T has been done, as advised in the problem formulation, by starting it at 1.0 and decreasing it with 0.05. This test was carried out only on the fast marker motion since here it would have

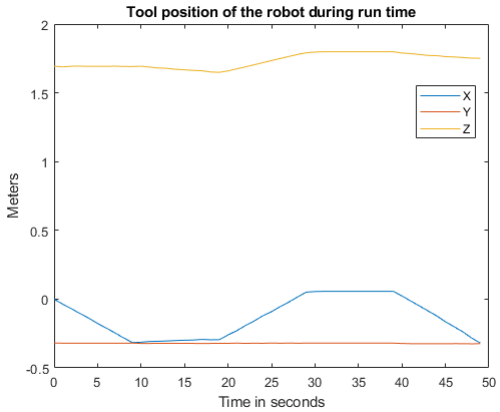


(a) Tool position

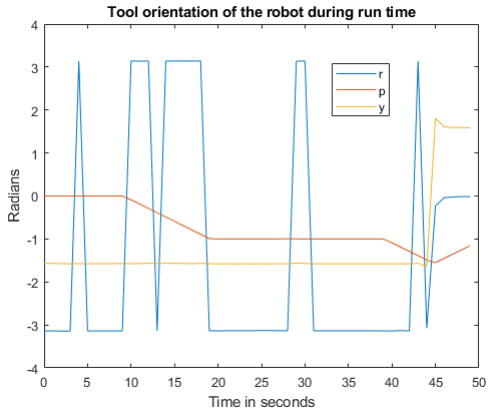


(b) Tool orientation

Figure 6: The tool pose of the robot when tracking a single point during a run with a delta_T of 1 second on the fast marker motion.



(a) Tool position



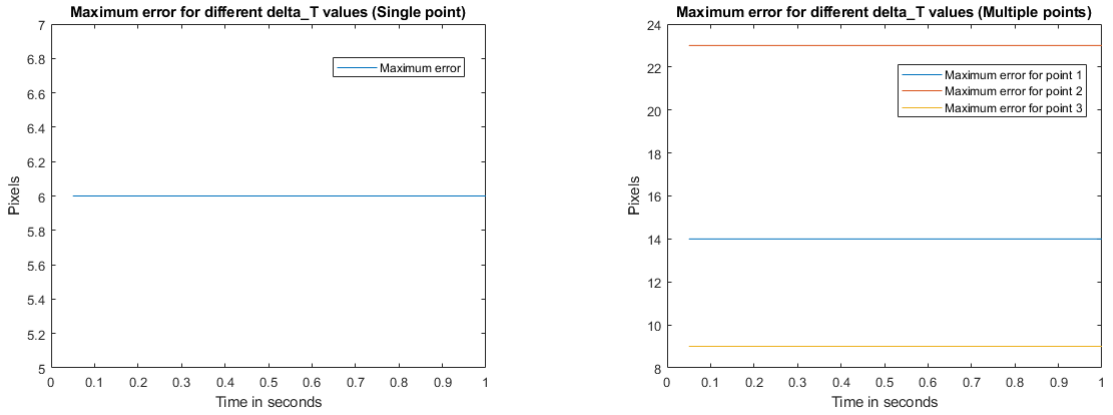
(b) Tool orientation

Figure 7: The tool pose of the robot when tracking three points during a run with a delta_T of 1 second on the fast marker motion.

the largest impact. For each run the max error has been found and plotted against the delta_T in figure 8a. This value is constant for a single point, which was expected since the velocity limit is never broken, even at the lowest delta_T. For multiple points the values are also constant for the same reason as when tracking a single point. This can be seen on figure 8b.

4 Combining Feature Extraction And Tracking

Qt related implementations will not be discussed, since that is not part of the scope of this project. Likewise RobWorkStudio and plugin interfacing related topics are not discussed as well.



(a) Maximum error for when tracking a single point

(b) Maximum errors when tracking multiple points

Figure 8: Maximum tracking error(s) when running on the fast marker motion.

4.1 Using The Plugin

On figure 9 an image of the plugin is seen. Clicking on the *Init* button will initialize the plugin and reset the WorkCell. No marker is active as well as no background when initializing the plugin. Therefore using tracking functionality in the plugin without choosing any marker is strongly discouraged. The user can click the *Select Background* button to choose a background and load it into the WorkCell. Likewise a user can click the *Select Motion* button to select a motion sequence to be loaded. It is very important that the motion sequence has the same format as the ones provided with this project. Clicking on marker 1 or marker 3 will setup the WorkCell to track that marker. Marker 2 has not been implemented, so that button does nothing when pressed. The user can select a delta_T that the plugin should execute with. If none is set, the default value is 0.5. Ticking the *active* tick box will start the tracking sequence with the chosen marker and delta_T. When the sequence is over the user should manually de-tick the tick box to stop the plugin from executing. The tick box can also be used as a pause button during the sequence. The video feedback in the plugin is the viewpoint of the robot.

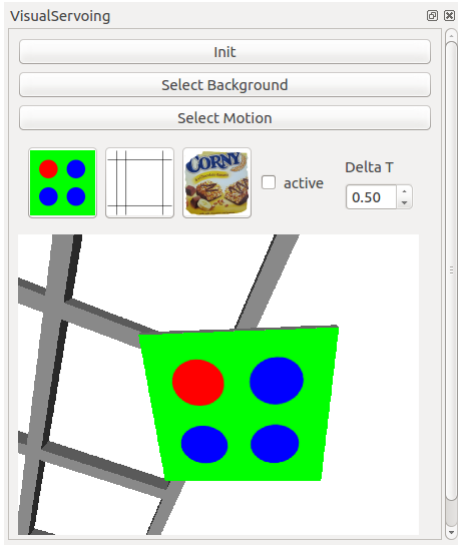


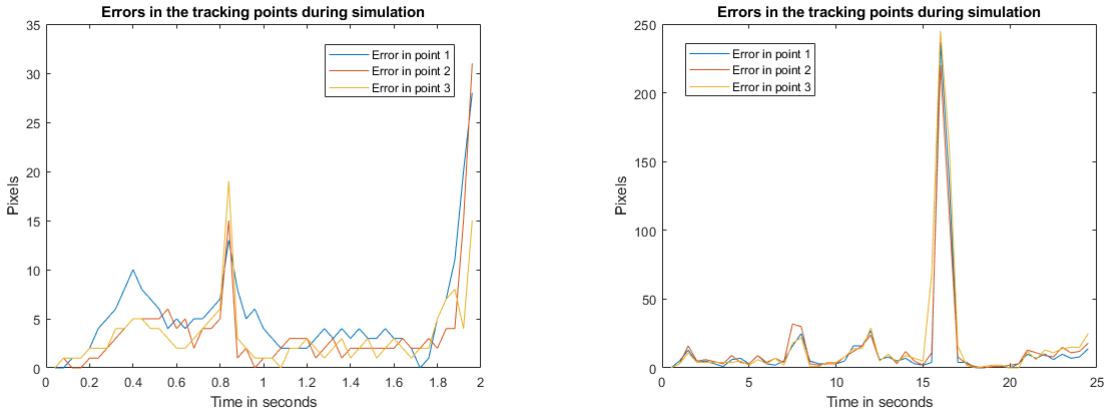
Figure 9: A picture of how the plugin looks.

4.2 The Plugin Implementation

Most of the functionalities and methods discussed in this has been collecting into a feature extraction and visual servoing library called respectively *ip*(image processing) and *vs*(visual servoing). This made the plugin easier to manage and control, since when a desired functionality from either library is wanted it is simply called. The plugin is setup with a Qtimer dependent on delta_T chosen by the user. Whenever this timer timouts the *nextMarkerPosMult()* function is called. This function is the bread and butter of the plugin, it iterates through the motion sequence chosen and updates the WorkCell accordingly. It should be noted that the plugin takes into account the image processing time estimated (in the project defined as τ) to the average runtime for either marker found in section 2.2. That is, the robot has that much less time to move in. *nextMarkerPosMult()* then uses the *ip* and *vs* libraries to find the marker reference points and then calculate a desired change to the robot configurations to track the marker. This is reflected in an update of the workcell.

4.3 Visual Servoing Test With The Plugin

Tests has been run on the plugin combining the feature extraction and visual servoing. The tests consists of tracking both markers with the fast motion sequence. On figure 10 the two plots of either markers shows the tracking error in pixels. In the figure, the tracking error for marker 1 seem to generally be smaller than the tracking error of marker 3, which indicates that marker 1 has a more stable motion. The two figures in figure 11 shows the configuration of the robot tracking both markers. These plots also indicate that marker 1 has a more stable motion compared to marker 3. On figure 12 the velocities of the robot joins are shown for when tracking the two markers. The velocities of the joints changes a lot more on marker 3 than on marker 1, which is again an indication of a more unstable motion while tracking marker 3 compared to marker 1.



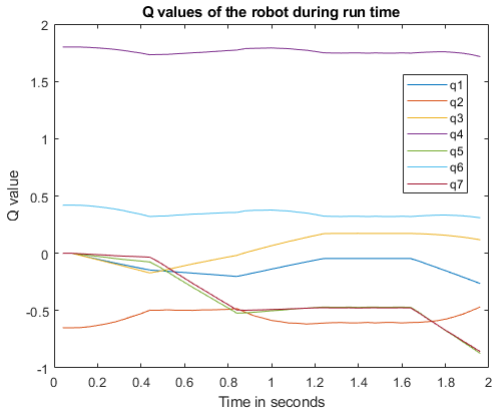
(a) Tracking error of marker 1

(b) Tracking error of marker 3

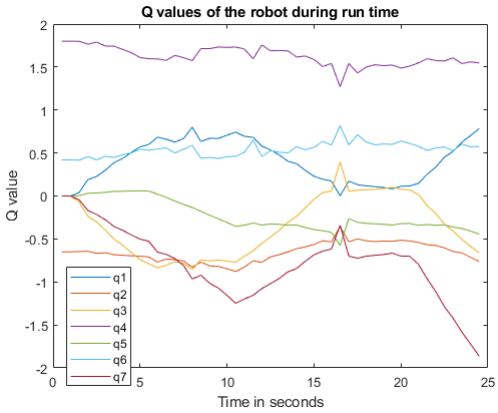
Figure 10: Tracking error of the robot during a run in the plugin. For marker 1 the delta_T was set to be 0.04 seconds and for marker 2 it was set to 0.5 seconds.

5 Concluding Remarks

First of all, we successfully tracked both marker 1 and marker 3 using the afore mentioned methods. When the robot is tracking marker 3 it might be more perpendicular to the marker, but it is also very "shaky" when tracking, whereas when the robot tracks marker 1 the tracking feels more smooth, but when marker 1 changes perspective the robot rather seem to move closer to the marker than trying to stay perpendicular to it. This could have been improved by increasing the precision of the found points in the vision related part. For the

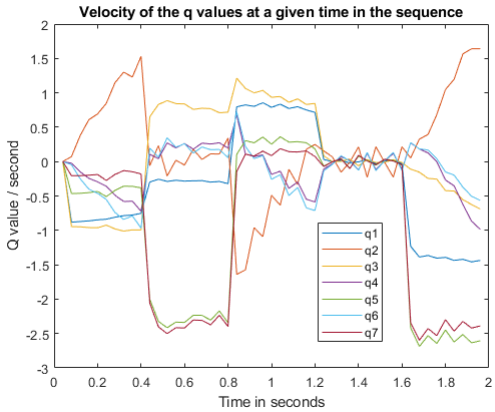


(a) Q values for marker 1

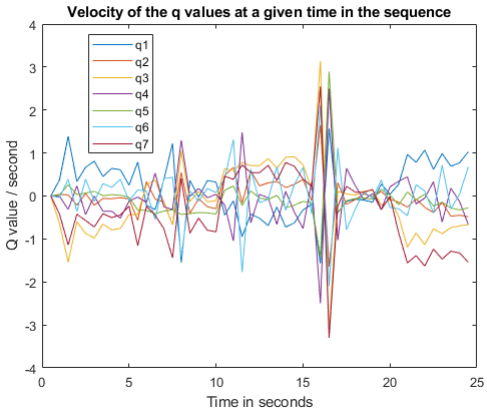


(b) Q values for marker3

Figure 11: Q values of the robot during a run in the plugin. For marker 1 the delta_T was set to be 0.04 seconds and for marker 2 it was set to 0.5 seconds.



(a) Joint velocities of marker 1



(b) Joint velocities of marker 3

Figure 12: Joint velocities of the robot during a run in the plugin. For marker 1 the delta_T was set to be 0.04 seconds and for marker 2 it was set to 0.5 seconds.

robotics part, the precision of the tracking could be improved by making an additional correction while the next image is being processed.

References

[1] OpenCV camera calibration and 3d reconstruction. `https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html?highlight=findhomography#findhomography`. Accessed: 2017-12-17.

[2] OpenCV common interfaces of descriptor matchers. `https://docs.opencv.org/2.4/modules/features2d/doc/common_interfaces_of_descriptor_matchers.html?highlight=flannbasedmatcher#flannbasedmatcher`. Accessed: 2017-12-17.

[3] OpenCV feature2d module. 2d features framework. https://docs.opencv.org/3.0-beta/doc/tutorials/features2d/table_of_content_features2d/table_of_content_features2d.html#table-of-content-feature2d. Accessed: 2017-12-17.

[4] OpenCV features2d + homography to find a known object. https://docs.opencv.org/3.0-beta/doc/tutorials/features2d/feature_homography/feature_homography.html#feature-homography. Accessed: 2017-12-17.

[5] OpenCV operations on arrays. https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html?highlight=perspectivetransform#perspectivetransform. Accessed: 2017-12-17.

[6] Tinne Tuytelaars Luc Van Gool Herbert Bay, Andreas Ess. Surf: Speeded up robust features. *Computer Vision and Image Understanding (CVIU)*, 110(3):346–359, 2008.