University of Southern Denmark
Faculty of Engineering

Introduction to Robotics and Computer Vision

Final report

Authors:
Anders Bjørk
Malthe Høj-Sunesen

Supervisors:
Henrik Gordon Pedersen
Dirk Kraft

December 18, 2015

1 Introduction

The report is the result of the final project in Introduction to Robotics and Computer Vision. The project is about visual servoing. For the vision part we chose marker 1 and marker 3; see Figure 1. We chose these markers since we found the challenges interesting. The two markers also has a good angle for different solutions. Although SIFT/SURF could be used to identify the circles, a somewhat more intuitive function is implemented.

And now, somethingwomething about what we will try to show in the report.

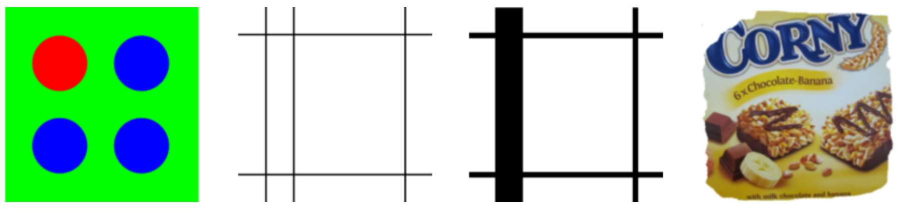


Figure 1: The four markers; 1, 2a, 2b and 3.

2 Vision

2.1 Tracking the circles

In order to track the circles it is necessary to find them first. A flowchart over the preparation and circle identification process can be seen in Figure 3 on the next page, while examples during processing can be seen in Figure 4 on page 4. In order to find the circles the image is first converted to a grayscale image. Next, it is put through the OpenCV function `adaptiveThreshold` in order to get a binary image. By using `adaptiveThreshold` it is the difference between the pixels in a neighborhood that leeds to the thresholding, rather than having to use color segmentation. It is not clear if this method is actually faster than channel-based thresholding. But it works.

After thresholding, the image is first `eroded` then `dilated` (opened) in order to remove noise and to separate the circles from the border of the plate. Then, the image is thresholded, as the adaptive threshold actually offsets the expected white color into a more gray color.

After the preparation, `findContours` is called. The contours are first filtered after area. Then a ratio, calculated after the formula in Equation 1 as explained by Henrik Skov Midtiby as a good metric for finding circles, is calculated.

$$ratio = \frac{4 \times \pi \times area}{perimeter^2} \tag{1}$$

This ratio is used to sort the contours in ascending order; lower ratios before higher ratios. A perfect circle (which is not possible in a picture due to the quadrilateral nature of pixels) would have a ratio of 1. This is easily shown by inserting the formulae for perimeter and area of a circle in Equation 1:

$$\begin{aligned} ratio_{circle} &= \frac{4 \times \pi \times (\pi \times r^2)}{(2 \times \pi \times r)^2} \\ &= \frac{4 \times \pi^2 \times r^2}{4 \times \pi^2 \times r^2} = 1 \end{aligned} \tag{2}$$

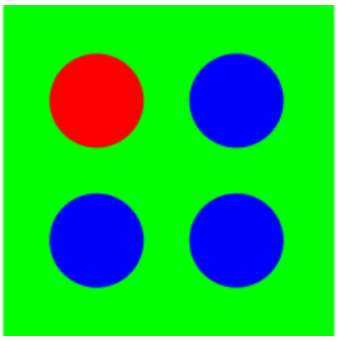


Figure 2: The circle marker used for tracking

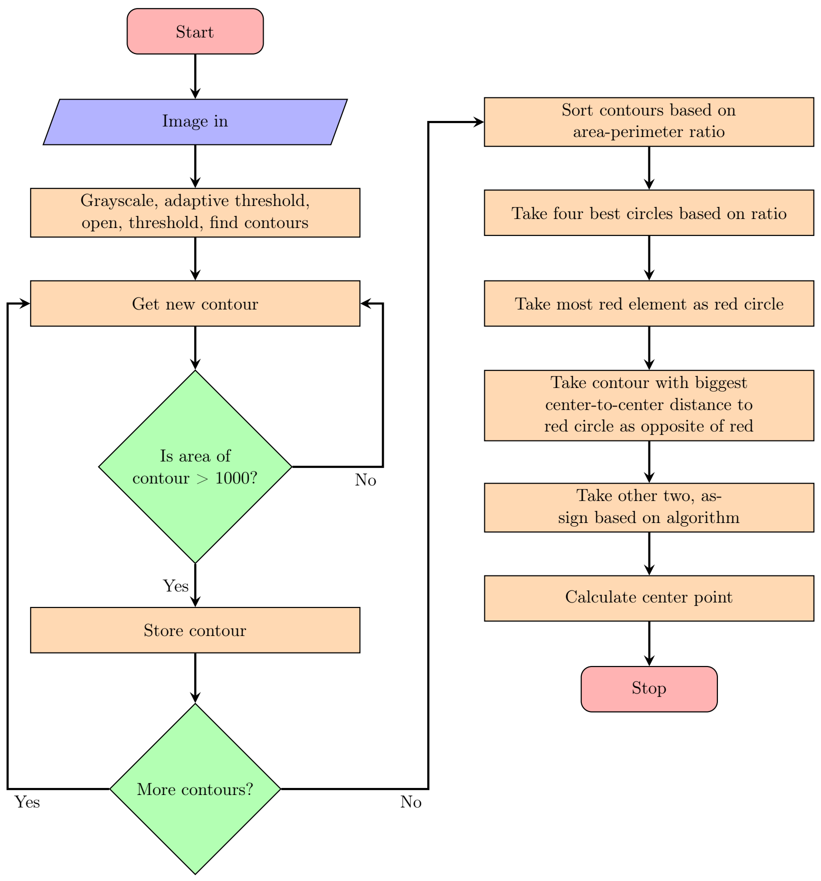
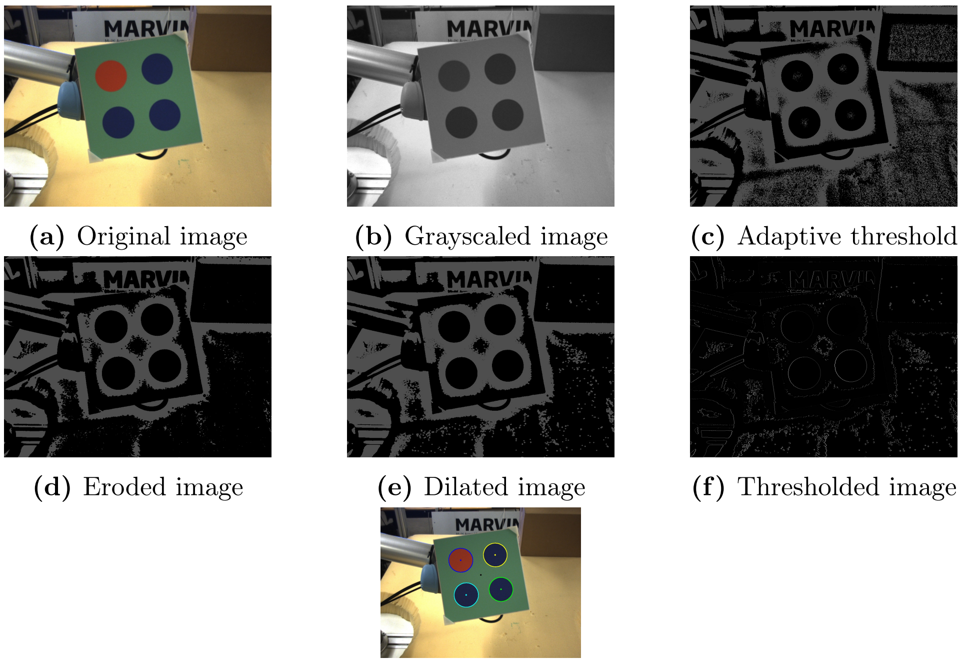

Start
Image in
Grayscale, adaptive threshold, open, threshold, find contours
Get new contour
Is area of contour > 1000?
No
Yes
Store contour
More contours?
Yes
No
Sort contours based on area-perimeter ratio
Take four best circles based on ratio
Take most red element as red circle
Take contour with biggest center-to-center distance to red circle as opposite of red
Take other two, as- sign based on algorithm
Calculate center point
Stop

Figure 3: Loading of image and detection and identification of circles.



(a) Original image **(b)** Grayscaled image **(c)** Adaptive threshold

(d) Eroded image **(e)** Dilated image **(f)** Thresholded image

(g) Found circles and center of marker (dark spot in the middle)

Figure 4: Image preparation process

Since a higher ratio means more circle-like, the four contours most like a circle is in the four last positions. It is assumed that the circles are in these four positions, which they indeed are in all test images in both the easy and hard sequence.

Using a contour's moment, it is easy to calculate the center of mass for the contour. This, since it is a circle, is the center of the circle. An average of the four centers is the midpoint between the four circles and thus the middle of the marker.

The developed identification routine singles out each of the four circles and uniquely identifies them based on their position in the original marker (see Figure 2 on page 2); red, top right, bottom left, bottom right. The identification is as such not used for e.g. extracting pose or "orientation".

Finding the red circle is a matter of comparing the color of the four found circles. The circle which has a red value higher than the other three is the red circle. In this implementation, only a single pixel (the one in the middle) is used; a better implementation would be taking an average over a larger sample. The circle opposite of the red circle will be the circle to which red has the highest distance.

Those were the program linewise quick ones.

Finding the last two circles are based on four scenarios, as seen in Figure 5 on the next page. First, the scenario must be deduced from the x and y coordinates of the red and opposite circles. Then, the coordinate of one of the circles must be compared to the coordinate of one or both of the identified circles. Lastly, the two circles are identified.

The whole process of loading an image storage, preparing image, searching for circles,

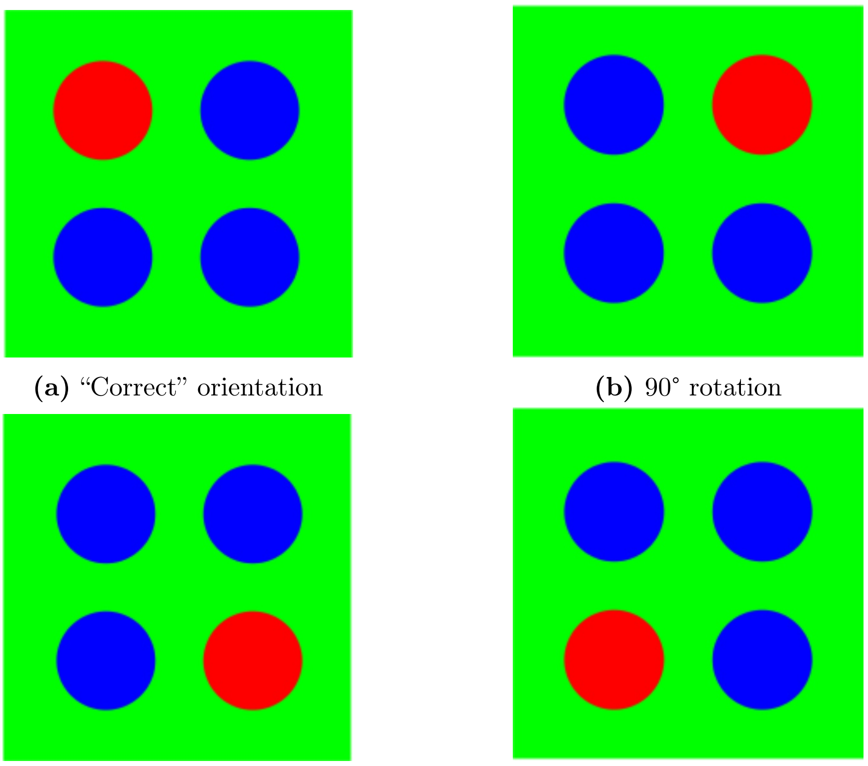

(a) "Correct" orientation **(b)** 90° rotation **(c)** 180° rotation **(d)** 270° rotation

Figure 5: Circle marker rotation cases

identifying them, finding midpoint of marker and returning that point can be done at > 30 fps. The test machine is an Asus X53Sv laptop with an Intel Core i7–2630QM CPU @ 2.00GHz processor, 8 GB RAM and a Samsung 850 EVO SSD. Testing was performed immediately after a reboot. The program correctly identifies the circles in every image.

2.2 RANSAC and SURF

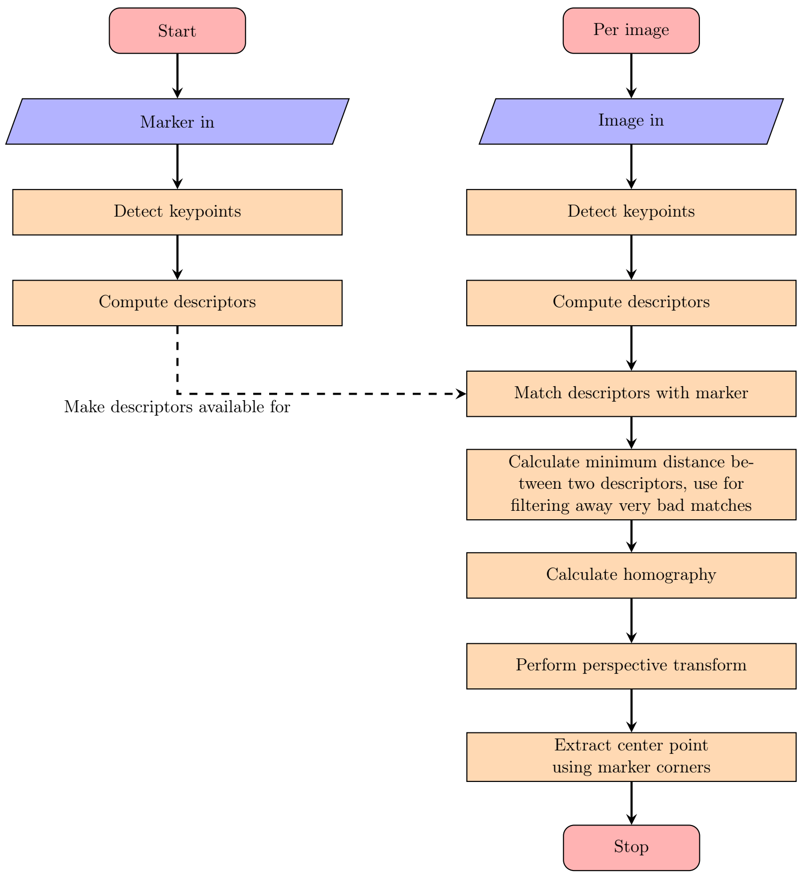

Start
Marker in
Detect keypoints
Compute descriptors
Make descriptors available for
Per image
Image in
Detect keypoints
Compute descriptors
Match descriptors with marker
Calculate minimum distance between two descriptors, use for filtering away very bad matches
Calculate homography
Perform perspective transform
Extract center point using marker corners
Stop

Figure 6: Loading of image and finding the corny marker and center.

Speeded up robust features, SURF, is faster than SIFT according to its authors[2], thus it is better in a time-critical implementation. It is also claimed to be more robust that SIFT, and so it is a better choice than SIFT. There appear to be no difference in how to implement either in OpenCV. The process for finding the marker in an image is shown in Figure 6 on the preceding page.

First, when the object of the `Ransac` class is constructed the Corny marker is loaded and preliminary computations are done. Then, when each new image is first assigned to the object and the center is extracted, the following happens:

1. Find keypoints are found and descriptors computed
2. Match descriptors for the true marker and the image using Flann matcherd
3. Calculate distance between matches
    (a) Throw away all matches which are less than three times minimum distance apart
4. Get coordinates for good matches
5. Calculate homography based on coordinates
6. Find corners in true marker
7. Perform perspective transform
8. Find corners in marker in image
9. Find center of the image using the average of the corners

Using that approach, the center of the image as well as the four corners are available.

Tests showed that this method is somewhat slow. On the same test machine as earlier, the metod could reach 6 fps with the easy sequence and barely 3 fps with the hard sequence. SURF/RANSAC is obviously not a fast way to extract features, although it can handle vastly more complicated markers than circle detection.

The RANSAC/SURF method finds the marker in both the easy and the hard sequence. The center point is not in the same position in all images, but is within 20 pixels. For tracking purposes this is sufficient; for interaction purposes (i.e. real world implementation) it might not be. When parts of the image goes outside the camera frame, the problem gets more pronounced.

It should now be clear that even though the two feature extractors are very different in their construction and method, they come to much the same result; four points in a quadrilateral and a point in the center of mass. Both methods are called the same way; instantiate the class, `assign(Mat)` an image and `extract()` the features wanted.

To extract the center point: `Point2f p = class.extract();`. To extract three points: `Point2f p1, p2, p3; class.extract(p1, p2, p3);`.

3 Robotics

3.1 Inverse kinematics

In order to use the image information, it is necessary to perform inverse kinematics. The end result should be

$$\mathbf{dq} = \left[\mathbf{Z}_{image}(\mathbf{q})\right]^T \mathbf{y} \tag{3}$$

Going backwards, $\mathbf{Z}_{image}(\mathbf{q}) = \mathbf{J}_{image}\mathbf{S}(\mathbf{q})\mathbf{J}(\mathbf{q})$. Here,

$$\mathbf{J}_{image} = \begin{bmatrix} -\frac{f}{z} & 0 & \frac{u}{z} & \frac{uv}{f} & -\frac{f^2+u^2}{f} & v \\ 0 & -\frac{f}{z} & \frac{v}{z} & \frac{f^2+v^2}{f} & -\frac{uv}{f} & -u \end{bmatrix} \tag{4}$$

$$\mathbf{S}(\mathbf{q}) = \begin{bmatrix} \left[\mathbf{R}_{base}^{tool}(\mathbf{q})\right]^T & | & 0 \\ ----- & & ---- \\ 0 & | & \left[\mathbf{R}_{base}^{tool}(\mathbf{q})\right]^T \end{bmatrix} \tag{5}$$

$$\mathbf{J}(\mathbf{q}) = \begin{bmatrix} \mathbf{A}(\mathbf{q}) \\ ------- \\ \mathbf{B}(\mathbf{q}) \end{bmatrix} \tag{6}$$

where, for Equation 4, f is the camera focal length and

$$\begin{aligned} u &= \frac{fx}{z} \\ v &= \frac{fy}{z} \end{aligned} \tag{7}$$

It is important to note that the x and y values do not correspond to pixel values from OpenCV, but must be offset with respectively half image width and half image height.

Equations 5 and 6 are further explored in [1]. Suffice to say, they are based on the robot's position $\mathbf{p}$ and state vector $\mathbf{q}$ among others.

4 Conclusion

The implemented visual feature extractors worked in every single test case. The circle marker is identified down to individual circles, and a total of five unique points are known. This is done faster than 30 fps or faster than 34 ms per frame, which is reasonably fast. Finding the corny marker takes a while longer; at 6 fps (167 ms per frame) for the easy sequence and just below 3 fps (333 ms) for the hard sequence, the method is not suited for time-sensitive implementations.

However, due to health problems relating one of the group members, the actual visual servoing was not implemented.

References

[1] Class material *Robotics introduction book*. Available on Blackboard.

[2] Wikipedia. *Speeded up robust features.* `https://en.wikipedia.org/wiki/Speeded_up_robust_features`. Accessed 2015–12–18.