Intelligent Multimedia Systems
Mean-Shift Tracker

Authors:

Agnes van Belle *(10363130)*,
Norbert Heijne *(10357769)*

December 27, 2012

1 Introduction

This paper is written for the course Intelligent Multimedia Systems (IMS) at the UvA as part of the standard curriculum of the Master Artificial Intelligence. For this assignment the goal is to implement and compare the mean-shift object tracker to a baseline object tracker, in our case the brute-force sliding window approach.

Object tracking is not a trivial matter, the computational cost for a brute-force approach is usually too high when it comes to intensive searching. The computational cost of object tracking in general is not the only problem this field faces. Many image and object conditions influence how well an object can be tracked and which strategy should be implemented.

Image condition can include the displacement or scaling of the entire image by moving the camera, this can result in a much larger displacement of the object relative to its previous position or requiring a differently scaled search window to properly classify the object. Changes in lighting conditions can cause colors and/or intensity to change, which could cause trouble depending on the color model that is used.

Some attributes of the tracked object can cause problems as well. If the object moves very fast, or if the framerate is low, the displacement can be a problem for some algorithms, such as mean-shift. An image cluttered with objects that are similar could cause the tracker to generate false positives, and lose our initial target in the process. Any change to the object in respect to color, intensity, scale, shape, rotation, or by occlusion can cause the algorithm to lose track of its target depending on the tracking method used.

Some commonly used tracking methods are: Brute force tracking; motion segmentation based tracking; template tracking; mean-shift tracking; Kalman filtering and particle filtering [3]. For this assignment we implemented the brute force tracker and the mean-shift tracker [2].

To compare the the mean-shift object tracker to a brute-force sliding window object tracker, the computation time and the accuracy of the tracker is considered. Using this comparison we will draw a conclusion on which tracker performs best in terms of time and accuracy under different image and object conditions. We expect the mean-shift to perform better with respect to computation time in every case, but we expect the brute force algorithm have a greater accuracy.

2 Trackers

Most trackers rely on a descriptor of an object and comparing candidates to the target descriptor. In our case the descriptor of the target is a 3d histogram of the RGB values or a 2d histogram of the normalized-rgb

values, the resulting histogram is then compared to the histograms of every candidate located by the tracker. This is called template matching, where the template is the target descriptor.

Both of the trackers described below compare histograms of RGB values and normalized-rgb values. To determine the similarity between the histograms, the Hellinger distance that incorporates the Bhattacharyya coefficient is used as the metric. The reason we use the Hellinger distance is because we use our histogram as a probability density function; the frequencies are normalized. This makes our distribution continuous and therefore we need to use the Hellinger distance to calculate the metric.

The Bhattacharyya coefficient for continuous probability distributions is calculated as followed:

$$BC(a,b) = \sum_{u=1}^{n} \sqrt{\pi(a_u) \cdot \pi(b_u)}$$

Where a and b are the target and candidate histograms, and $\pi_i(a_u)$, $\pi_i(b_u)$ are the normalized frequencies of members of the histograms a and b in the uth bin.

The Hellinger distance is then calculated as followed:

$$H(a,b) = \sqrt{1 - BC(a,b)}$$

2.1 Brute force Tracker

The brute force tracker uses a sliding window approach, this means that a search window slides over the image and compares the target descriptor to the candidate descriptor. Sliding over the image means starting at a corner in an image, creating a candidate, calculate the similarity between the candidate and the target, storing the results and then moving the search window with a set amount of pixels. Every possible position of the search window is considered a candidate. This is a very slow way of finding the best candidate for the target and makes the computational cost high. The computational cost can be lowered using a few heuristics and programming tricks, more on this in section 3.1 where we discuss the implementation of the brute force tracker.

2.2 Mean-shift Tracker

For this tracker the Bhattacharyya Coefficient $\rho[\hat{p}(\hat{y}_0, \hat{q})]$ Maximization [2] is used to find the coordinates for which the similarity between the target histogram and the candidate histogram is highest.

In more detail the mean-shift tracker creates a candidate histogram of the last known location of the target, but instead of the Hellinger distance the candidate is now evaluated using a gradient. To calculate the gradient, weights are applied to each pixel's distance to the center of the search window. The weights are calculated by comparing the bins of the target and candidate histogram to which that pixel belongs.

$$w_i = \sqrt{\frac{\pi_i(a_u)}{\pi_i(b_u)}} \qquad\qquad i \in u$$

Where a and b are the target and candidate histograms, and $\pi(a_u)$, $\pi(b_u)$ are the normalized frequencies of members of the histograms a and b in the uth bin so that pixel i is a member of the uth bin.

The weighted average of these distances gives us a gradient to follow. Note that this only gives a good gradient if the last known position of the target still has some overlap with the true position of the target. If the object moves too fast, if the framerate of the video is too low or if the camera has moved significantly then other methods have to be used to help the mean-shift tracker find the target again.

The mean-shift tracker also uses a kernel overlay to adjust the weights of the pixels in the search window before computing the histogram. For our tracker we used the Epechnikov kernel function, this is roughly ellipse shaped with the edges of the search window as boundaries, making the weights of the pixels in the corners of the search window zero and gradually higher towards the middle.

3 Implementation

In the following section the implementation of both trackers will be explained in further detail. For the most part adjustments and improvements will be discussed. Both trackers have been implemented using Matlab R2011b.

For both the trackers the RGB and normalized-rgb color models have been used. The use of normalized-rgb implies the use of a 2d histogram, but we chose to use the standard 3d histogram, because the expected benefit of using a 2d histogram is not that large. We have decided to use the normalized-rgb color model to adjust for the intensity changes from frame to frame, though we expect to benefit from this, it is most likely not going to influence the results much, since most of these slight intensity changes have no effect because of smoothing.

Some things have been left out and will be discussed later on in sections 7 and 8. Among these are ways of handling occlusion, changes to color, scaling and rotation.

3.1 Brute-force Tracker Implementation

In this section we will discuss the implementation of the brute force tracker. Since the brute force tracker is so computationally expensive, a few tricks have been used to speed up the process.

First we apply a heuristic to the search space, for this we make the assumption that the object that we're tracking does not have a large displacement from frame to frame. This means that the object is not very fast or that the framerate is high enough so that the displacement is small. The search space is pre-defined in our algorithm as having two times the width and height of the target in pixels, with the last known position of the target as the center. Searching through the entire image is not needed to do single object tracking, so this is an obvious step forward.

The following methods are used to either speed up the program or are hacks to increase accuracy. Some are used and some are stated just for reference.

We can also reduce the amount of bins in our 3d histogram to 5^3, this is actually more of a smoothing method, but it also reduces the amount of calculations needed for the comparison between the target and candidate histogram.

The prominent reason why the brute force tracker is slow is that every candidate histogram has to be calculated, even though the candidate histograms have a large overlap. Therefore using the previously calculated histogram to recalculate the following histogram is a large improvement. The larger the search window, the more this method decreases the amount of time needed to calculate the histograms.

A second method we could use to reduce the amount of time needed per frame is to skip pixels and therefore skipping candidates. Instead of moving the search window one pixel at a time, we move the search window three pixels and then calculate a new candidate histogram. Of course the more pixels we skip the more it will hurt the accuracy of the tracker, though this still seemed like a minimal drop in accuracy compared to the reduction in the computational cost per frame. For the baseline however we will not use this speedup since we want the results to be as clear as possible.

We also apply the same kernel function the mean-shift tracker uses to the brute force histograms to enhance our results and to reduce the influence of the background pixels that are more prominent towards the edge of the search window when focusing on an object.

3.2 Mean-shift Tracker Implementation

In this section we will discuss the implementation of the mean-shift tracker. Mean-shift is very fast in comparison to the brute force tracker, the only problem is that once the tracker completely loses track of the target, it needs help in finding it again. This has not been implemented but is discussed in section 8.

The implementation was done according to the paper by [2], we have made two versions however and each with an adjustment. The first version (MS1) has the following adjustment: When computing step five of the Bhattacharyya Coefficient Maximization, if no point can be found for which the distance is better than the last known location then it takes the second best out of every point that has been evaluated, we

have set our limit to consider 5 halfway points. This is to avoid the obvious loop in the algorithm that occurs if we exclude the last known position as a valid option and still wish for a better point. The second version (MS2) has the following adjustment: Also when computing step five of the Bhattacharyya Coefficient Maximization, instead of checking whether the new found point is better than the last known location, we take the halfway point (gradient divided by 2). Thereby skipping a step in the algorithm. This is a minor speed up with no expected loss of accuracy.

4 Results

We have examined the difference on two data sets. One data set, which we were given, contains a soccer scene of 201 frames. With regard to this data set we have averaged over two different soccer players that we have tracked. The first soccer player was the orange one in the upper left corner in the first frame. The second player is also an orange player but the one in the upper part of the circle in the first frame. See also Figure 1.

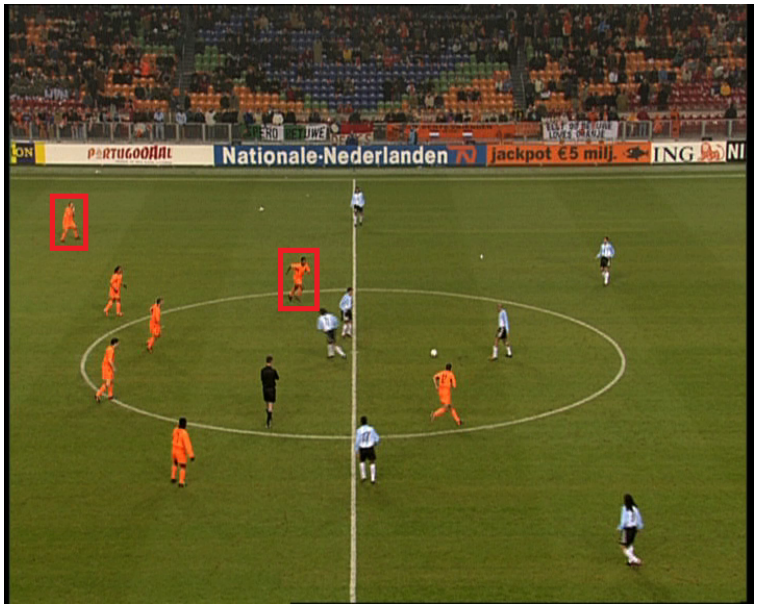

Figure 1: The two tracked players from the soccer data set

Another data set that we use is based on an animated movie of a Kiwi bird [1]. We have taken two scenes from this movie, one that includes many rotations and movements, S_1, and another that includes an obvious occlusion, S_2.

We have compared between the algorithms (the Brute-force tracker and our two version of the Mean-shift tracker as described in Section 3.2) on the soccer set. The results are in Figure 2. Overall there is no clear decision between the MS1 and MS2 tracker. It seems a trade-off between time and accuracy. We would expect the tracker MS1 to have a better accuracy in term of its distance to the target histogram, because it always checks if the position half way between the found location (with mean-shift) and the current position is better.

We have also compared between color spaces using the various trackers. From Table 1 we see that the normalized rgb color space yields a relative high improvement trading off for a slight increase in average time. This is the same pattern for all trackers.

[1] `http://www.youtube.com/watch?v=sdUUx5FdySs`

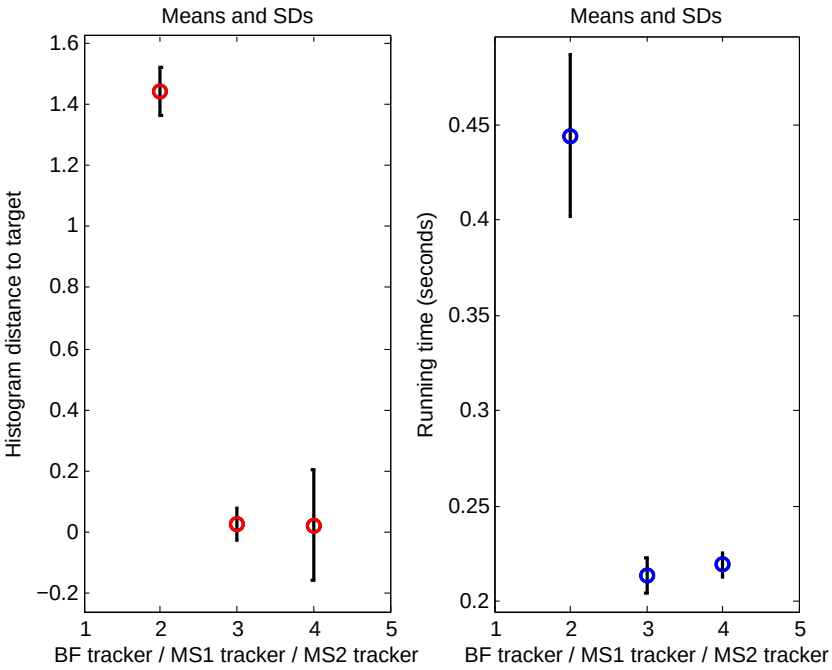

Figure 2: Means and SD's for performance measures (averaged over frames) for the 3 trackers for the soccer data set

In Tables 2 and 3 we show the results for the Kiwi data set from which we have taken two different scenes to measure specifically for rotation and occlusion. The concept that appears from these data is that a higher frame rate benefits accuracy (target histogram distance) relatively much, while the average extra time cost is a number that starts at two digits after the comma. This is probably because there is extra time needed to compute the new location when the frame rate is low, especially so in these cases, with many fast movements and when occlusion occurs.

In Figure 3 we show example frames from the Kiwi occlusion image set S_2.

		Algorithm			
Color space	**Measure**	BF	MS1	MS2	**Means**
RGB	Hist. dist.	0.4977	0.2843	0.2789	0.3536
	Running time	1.4163	0.0262	0.0213	0.4879
Norm. RGB	Hist. dist.	0.3908	0.1429	0.1590	0.2309
	Running time	1.4690	0.0262	0.0220	0.5057

Table 1: Results (averaged over frames) considering the color space for the soccer data set

Avg. distance for tracker MS1 with norm. RGB		
	Scene type	
Nr. frames	Rotation and movements (S_1)	Occlusion (S_2)
few	0.133	0.2565
many (few $\times 6$)	0.1348	0.2477

Table 2: Results (averaged over frames) considering the distancte to the target histogram for the Kiwi data set

Avg. time for tracker MS1 with norm. RGB		
	Scene type	
Nr. frames	Rotation and movements (S_1)	Occlusion (S_2)
few	0.1631	0.9542
many (few $\times 6$)	0.0864	0.5315

Table 3: Results (averaged over frames) considering the time in seconds for the Kiwi data set

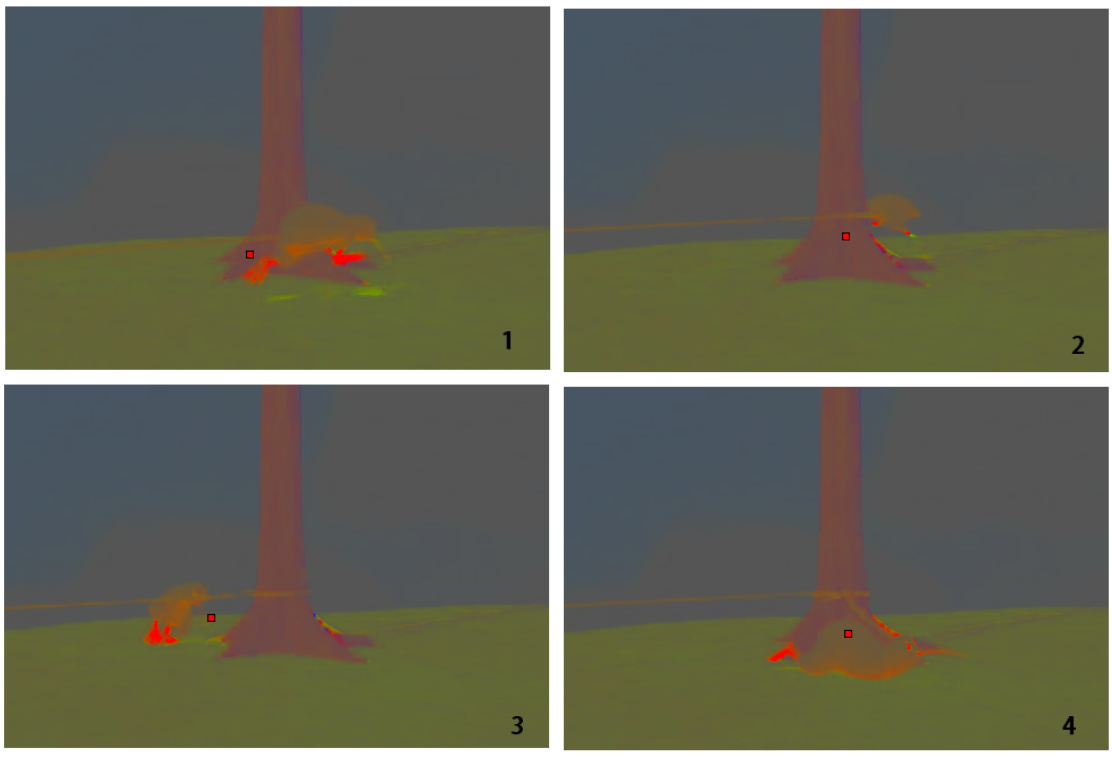

Figure 3: Example of dealing with occlusion in the Kiwi data set's occlusion scene (normalized RGB)

5 Analysis

The MS1 seems to perform better than the other trackers, mostly due to the fact that the tracker didn't lose sight of the target, always taking the second best location in the case of there note being a point along the gradient which is better than the last known location, seemed to have improved the tracking in case of partial occlusion and the brute-force trackers both had hiccups during tracking. MS2 had difficulties with sudden large displacements of the object, because of the way it was programmed. Taking the half-way point is a good way of not overshooting your target, but this lead to the loss of the target itself, probably because of the target transformation and the low resolution of the target. The brute-force tracker performed well enough, but was very slow compared to the mean-shift trackers.

6 Conclusion

The brute-force tracker and the mean-shift trackers MS1 and MS2 have been implemented for the sake of comparison. These two trackers have been compared with the color models RGB and normalized-rgb, with the accuracy of the candidates and the time that it took the tracker to finish each frame.

The results show that both the trackers performed better when normalized-rgb was used, MS1 performed slightly worse than MS2 in the average time needed per frame, but the histogram distance was slightly less than MS2. The brute-force tracker performed worse in all cases even though we would expected it to find the best possible histogram distance out of all the candidates.

With our results in mind the MS1 version had the least standard deviation as well, therefore it is safer to say that the MS1 performed better than the MS2. The brute-force tracker performed worse than both the mean-shift versions.

7 Discussion

Most of the trackers were able to track most of the targets, and only with certain conditions did the trackers lose track of their target. The MS1 seemed to keep track of the target, but the MS2 tracker seemed to lose their target when there is a sudden burst of speed or when occlusion occurs with the target. Possible adjustments that can be made to improve the results are found in section 8.

In case of scaling and occlusion the particle filter mentioned in [3] seemed to have a good balance between accuracy and computational cost, with current age computers, this would probably run well in real-time. A comparison between the mean-shift algorithm and the particle filter has been made in that paper, and given that the conditions for this data set are comparable to the examples shown in the paper, it is reasonable to assume that the particle filter would cover most of the faults of the mean-shift algorithm.

The mean-shift tracker is not robust in all domains, we have discussed some of these domains in this section. The mean-shift needs to be adapted in order to handle partial and complete occlusion and sudden large displacements. Once the tracker loses sight of the target, the non-adapted algorithm cannot cope with it.

8 Future Work

Two ideas for helping the tracker recover from losing the target have not been implemented. If the tracker loses track of the target we can notice this by looking at the individual pixel weights that have been computed as part of the gradient calculation. The gradient alone cannot tell us if we've lost our target or if the target is standing still. Therefore if all the pixel weights are below a certain threshold or if the calculated distance between the target histogram and the candidate histogram is too high we know that we've lost our target.

Therefore we suggest keeping track of the average displacement in distance traveled and averaged coordinate shift of the last k steps, and using random sampling to catch sight of the target. Keeping track of the average distance traveled means that we can estimate how fast the object travels each frame and therefore

the radius in which we can take random samples, and keeping track of the averaged coordinate shift means that we can estimate the direction the object traveled in. We combine these two to estimate a position, direction and average distance traveled, from which we take random samples in the vicinity with a radius of the average distance traveled. When a candidate has been found with a low enough distance to the target histogram then we take that candidate and compute the gradient again.

We expect that this would help in cases of complete occlusion and where the target is moving in a straight line. Also when the target moves too fast and the search window is too small, this method can be used to find the target again.

A more detailed description of the earlier mentioned method to reduce the computation time for the histograms in case of the brute-force tracker is found in the paper by [5], the idea of recalculating the next histogram based on the previous histogram is not hard to think of. They have however improved on it by using a specialized hash table for the removal and insertion of pixels in bins. Also they propose to calculate all the histograms for every candidate before calculating the distance. Covering most of the computational cost in one go. Using a specialized insertion and removal method can help significantly if dealing with large amounts of candidates. Some of the problems that object tracking faces that were discussed in the introduction have not yet been discussed. We will not go into detail about all of them, but we will try to propose some solutions to the following domains: color changes and scaling.

Color changes are hard to deal with if we're using a histogram as our descriptor, because if all the bins suddenly change, that would be considered the same as the object not being the same object as the initial target. Therefore we need a secondary descriptor which is as robust as the histogram. For this we propose the use of natural image statistics. One option is computing the distance between Weibull functions, then weighting both descriptors, depending on the situation one might want to give more weight to another descriptor. Another option is to use blob detection and use their frequency in the same way as the histograms and compute a gradient. Depending on the type of object and background, this could prove disastrous or efficient. If the blobs differ a lot when the angle of the object changes then this is not very useful, but if not, then a proper gradient can be computed.

Scaling relatively requires a lot more computation, usually a multitude in computational depending on how many scales are checked if done the brute-force way. This makes looking for an object that changes scale constantly much more intensive. The use of a prior to which we can estimate the scale beforehand would be very useful. To use this prior we have to make a few assumptions: When an object scales up or down, the change is always a fraction of the current scale. This fraction is constant for every frame if the direction of the object does not change.

In case of the mean-shift tracker and because of the second assumption this would mean that the direction can be used to estimate the scale if the scale has changed over the course of the last few frames. The problem is with the first assumption, this fraction is unknown because we do not know the distance from the camera to the object in reality. The only way with the mean-shift tracker to estimate whether or not the object is scaling is by interpreting the Bhattacharyaa Coefficient, if the coefficient is changing over time then it indicates that the object is changing. A shaky strategy at best, because we do not know in what way it is changing. Using a secondary descriptor such as edges could indicate scaling, using the same method as the mean-shift the pixel weights could indicate whether or not the frequency of edges has become higher or lower. Higher frequency of edges, but shorter edges indicates scaling down, lower frequency but longer edges indicates scaling up. This way a prior can be set on the pixel weights of the edges instead of calculating a multitude of histograms for many different scales.

The color models that have been considered in this paper were RGB and normalized-rgb, both standard and easy to understand. Though normalized-rgb did help in some situations, this is probably not the optimal color model to use. Based on the comparison made in [1] the l1l2l3 model seems to work best, having the most invariance and given that the color of the illumination does not change often. According to [4] however, l1l2l3 seems to produce worse results in the detection of visible shadows than c1c2c3. The latter model having most of the benefits except that it is not invariant to highlights. Expected that the object will keep the highlights over the course of tracking, then the mean-shift tracking method using histograms should suffice. Both l1l2l3 and c1c2c3 are compatible with the current 3d histogram comparison, using the HSI model would

have been a good choice as well if not for the incompatibility with the 3d histograms. Therefore using c1c2c3 would make an optimal choice if we were to incorporate any of the edge detection that was mentioned above.

References

[1] T. Gevers, WM Smeulders, et al. Color based object recognition. *Pattern recognition*, 32(3):453–464, 1999.

[2] P. Meer. Kernel-based object tracking. *IEEE Transactions on pattern analysis and machine intelligence*, 25(5), 2003.

[3] K. Nummiaro, E. Koller-Meier, and L. Van Gool. An adaptive color-based particle filter. *Image and Vision Computing*, 21(1):99–110, 2003.

[4] Y. Shan, F. Yang, and R. Wang. Color space selection for moving shadow elimination. In *Image and Graphics, 2007. ICIG 2007. Fourth International Conference on*, pages 496–501. IEEE, 2007.

[5] Y. Wei and L. Tao. Efficient histogram-based sliding window. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3003–3010. IEEE, 2010.