

One of the central questions in video processing is how to follow an object over time. Imagine you are designing a game where the position of the hand is used as a controller. What you need from your video processing software is then the position of the hand in each image and hence a temporal sequence of coordinates, see Table 9.1.

We can also illustrate this as a number of points in a coordinate system. If we connect these points we will have a curve through time, see Fig. 9.1. This curve is denoted the *trajectory* of the object.

The notion of a trajectory is not limited to the position of the object. We can generalize the concept and say that the object is represented by a so-called *state vector*, where each entry in the vector contains the value of a certain parameter at a particular time step. Besides position, such entries could be velocity, acceleration, size, shape, color etc. Formally we define tracking to be a matter of finding the trajectory of an object's state. This chapter will define a framework for tracking, namely the so-called *predict-match-update* framework, see Fig. 9.6. Without loss of generality we will below assume the state is only the position of the object, meaning that the state vector we seek to find is $\vec{s}(t) = [x(t), y(t)]$. Below the framework is built up one block at a time.

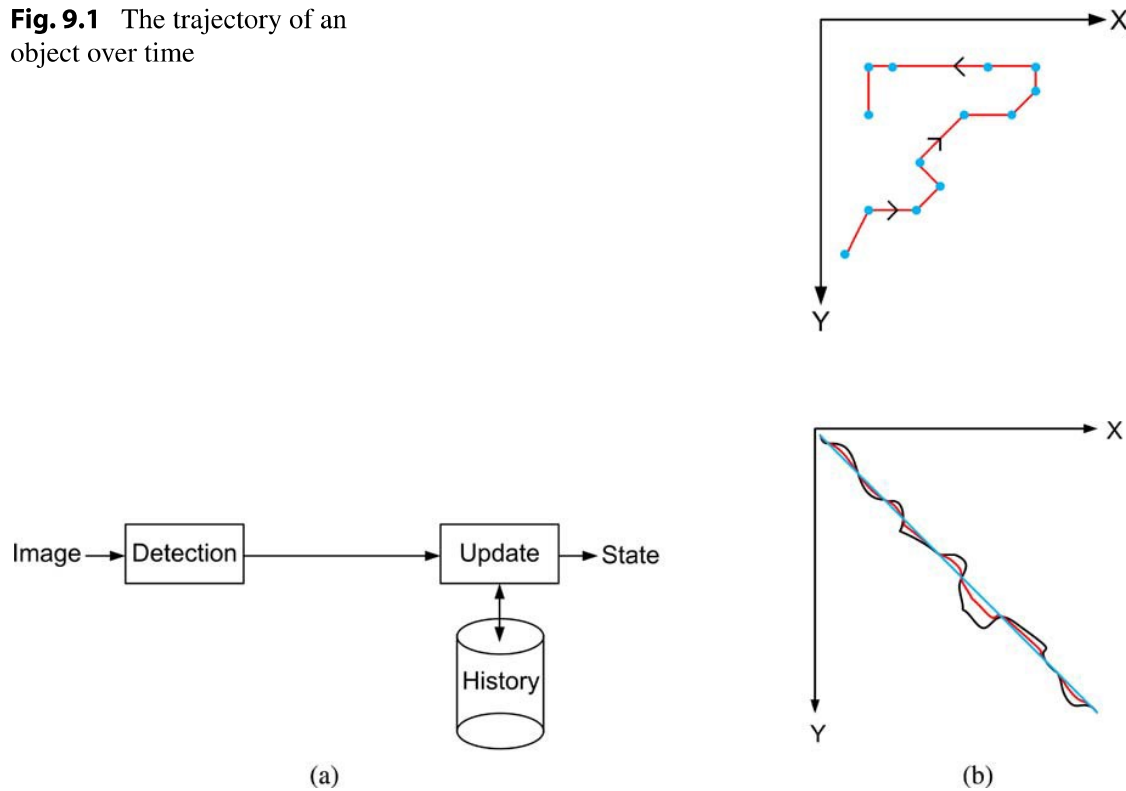
9.1 Tracking-by-Detection

We can use some of the methods described previously in the book to detect an object. If we do this in each image and simply concatenate the positions we could argue that we are doing tracking. This approach is, however, *not* considered tracking since each detection is done independently of all other detections, i.e., no temporal information is included.

The most simple form of tracking is when the estimated position is updated using previous states. The current and previous states are combined in order to smooth the current state. The need for this is motivated by the fact that noise will always appear in the estimated position.

Table 9.1 The position of an object over time

Time	1	2	3	4	5	6	7	8	9	10	11	12	13	...
X	1	2	4	5	4	6	8	9	9	7	3	2	2	...
Y	10	8	8	7	6	4	4	3	2	2	2	2	4	...

Fig. 9.1 The trajectory of an object over time**Fig. 9.2** (a) Framework for updating the state. (b) The effect of updating the states. The *blue curve* is the true trajectory of the object. The *black curve* is the detected trajectory and the *red curve* is the smoothed trajectory

Smoothing can be implemented by calculating the average of the last N states. The larger N is, the more smooth the trajectory will be. As N increases so does the latency in the system, meaning that the updated state will react slow to rapid position changes. For example if you are tracking a car that is accelerating hard or is doing an emergency break. This slow reaction can be counteracted by also including future states in the update of the current state, but such an approach will delay the output from the system. Whether this is acceptable or not depends on the application. Another way of counteracting the latency is to use a weighted smoothing filter. Instead of adding N positions together and dividing by N , we weight each position according to its age. So the current state has the highest weight, the second newest state has the second highest weight etc. No matter which smoothing method is used to update the state, it is a compromise between smoothness and latency. In Fig. 9.2 the updating of the state is illustrated. The history-block contains previous states.

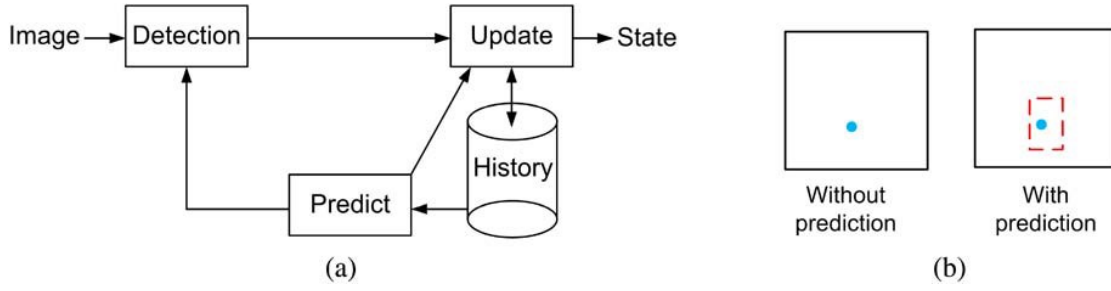


Fig. 9.3 (a) Framework for updating and predicting the state. (b) The effect of predicting the position of the object in the next image

9.2 Prediction

Very often the object we want to follow is moving much slower than the framerate of the camera. As a consequence the object is not moving very much from one image to the next. So, having located an object in one image will allow us to *predict* where the object will approximately be in the next image. We want to exploit this fact when detecting the object. This is done by introducing a ROI centered at the position where we predict the object to be and only analyze the pixels within the ROI, see Fig. 9.3. This will save a significant amount of processing time.

The question is now *where* we predict the object to be. For this purpose a *motion model* is introduced, that is, a model explaining how the object is moving. The most simple model is a *zeroth order linear motion model*. It predicts the object to be exactly at the same position in the next image as it is in the current image. The next of the linear motion models is the *first order linear motion model*, which includes the velocity of the object. Given the current position $\vec{p}(t) = [x(t), y(t)]$ and velocity $\vec{v}(t) = [v_x(t), v_y(t)]$ of the object, the predicted position will be

$$\vec{p}(t+1) = \vec{v}(t) \cdot \Delta t + \vec{p}(t) \quad (9.1)$$

where $\vec{p}(t+1)$ is the predicted position and Δt is the time between $\vec{p}(t)$ and $\vec{p}(t+1)$. Often the framerate is constant and Δt is simply the number of images predicted into the future. Usually we are just interested in predicting one image ahead and hence Δt can be removed from the equation.

The *second order linear motion model* also includes the current acceleration of the object $\vec{a}(t) = [a_x(t), a_y(t)]$ and the predicted position is given as

$$\vec{p}(t+1) = \frac{1}{2} \cdot \vec{a}(t) \cdot \Delta t^2 + \vec{v}(t) \cdot \Delta t + \vec{p}(t) \quad (9.2)$$

Again, with a fixed framerate and only predicting the next image, the two Δ terms become 1 and can therefore be ignored.

Motion models are not necessarily linear. If we for example are following an object being thrown, we need a model that includes gravity. Another example could be when tracking an object moving in a circle, the motion model would of course

be that of a circle, or if we are tracking a drunken human, the motion model might be more like a sinus curve than a straight line.

Sometimes the movement of an object cannot be explained by just one motion model. If we for example are tracking a fish in an aquarium, we will need two motion models. One model for when the fish is just swimming slowly around and another model for when the fish needs to get away from something fast. The first type of movement could be modeled by a first order linear model, while the other type of movement could be modeled by a random direction with maximum acceleration. Having two (or more) motion models will result in two (or more) ROIs.

No matter how good our motion model is, it is still just a model of the movement, i.e. an approximation. The object is therefore not likely to be located exactly at the predicted location. For this reason we look for the object within a ROI. The question is now how to define the size of the ROI. The simplest approach is to try different sizes and see which works best. A more scientific approach, which will render a smaller ROI and hence save processing time, is to define the size based on the *uncertainty* of the prediction. Say we in the last image predicted the x-position of the object to be at position 370, but found the object at position 350. Whether this difference is due to a bad prediction or a bad detection we do not know. What we *do* know is that there is some uncertainty in the x-direction. The bigger the uncertainty is, the bigger the ROI should be in the x-direction. Normally it is not recommended to let the difference control the ROI directly since it is sensitive to noise. A more conservative way of changing the ROI based on the difference is here shown for the width of the ROI:

$$\text{width}(t + 1) = \alpha \cdot |\underline{x}(t) - x(t)| + (1 - \alpha) \cdot \text{WIDTH} \quad (9.3)$$

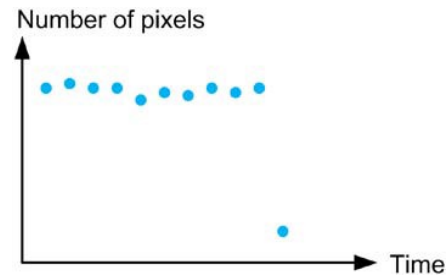
where α is a small value, $\underline{x}(t)$ is the predicted x-value of the object at time t , $x(t)$ is the detected x-value of the object at time t , and WIDTH is a predefined minimum width of the ROI. The same can of course be done in the vertical direction.

Similar to the uncertainty of the prediction, we also have an uncertainty associated with the detection. Imagine that we in one image have a bad segmentation of the object we are tracking. The effect of this could be that we only detect a small part of the object. We can still calculate the position of the object, but the number of object pixels used in this calculation is much smaller than in previous images, see Fig. 9.4. This would suggest that the detection has become more uncertain and ultimately we could have a situation where the object is not found and hence no detection is available. In both cases it might be better using the prediction than the detection when updating the state. Following along this line of thinking, the update of the state could be

$$\vec{s}(t) = \frac{w_1}{w_1 + w_2} \cdot \vec{p}(t) + \frac{w_2}{w_1 + w_2} \cdot \vec{p}(t) \quad (9.4)$$

where w_1 should be controlled by the uncertainty associated with the prediction and w_2 by the uncertainty associated with the detection.

Fig. 9.4 The number of object pixels as a function of time. Note how the number suddenly drops



Predicting is a delicate matter as we are talking about foreseeing the future. Care should therefore be taken before using any of the methods and equations presented above. But prediction in its simple form with a zeroth order or first order motion model, and a large ROI, is nearly always a good idea. So is the notion of including predicted values in the update when no detection is available.

9.3 Tracking Multiple Objects

Sometimes we need to track multiple objects at the same time. If the objects are different we can duplicate the methods mentioned above and track each object individually. When the objects are similar, however, we need a coherent framework that can simultaneously track multiple objects.

In the top row of Fig. 9.5 we see two similar objects that we want to track. Remember that tracking is about finding the trajectory of the object over time, meaning that we need to figure out which object is which in each image. This is known as a *data association* problem in the sense that we need to assign some data (here two detected objects) to their respective trajectories. In the figure it is obvious that we have an object to the left moving downwards while the object to the right moves upwards, but how does the computer infer this? The solution is to predict the ROI for each object, as discussed above, and investigate which of the two objects best *match* the respective ROIs. This is illustrated in the bottom row in Fig. 9.5. By including the matching block into the tracking framework, we have now arrived at its final structure, see Fig. 9.6. This tracking framework is denoted the *predict-match-update* framework.

Unfortunately, tracking of multiple objects is not always as simple as illustrated in Fig. 9.5. When objects move they are likely to occlude each other, which will result in objects disappearing or new objects appearing. Moreover, sometimes the

Fig. 9.5 Top row shows four consecutive images containing two moving objects. In the bottom row the *dashed red boxes* indicate the predicted ROIs. The numbers (#1 and #2) indicate which trajectory an object is found to belong to

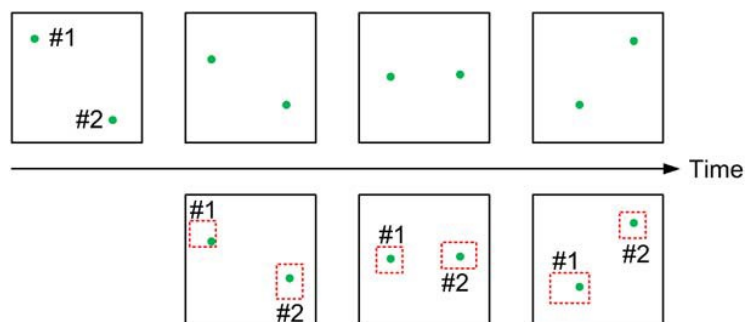


Fig. 9.6 The predict-match-update tracking framework

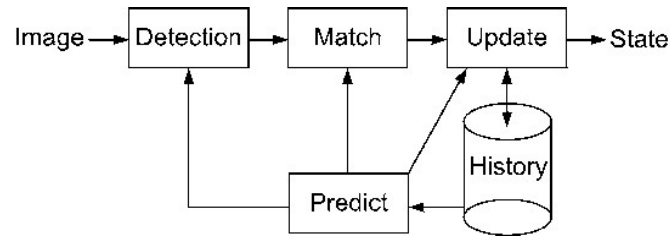
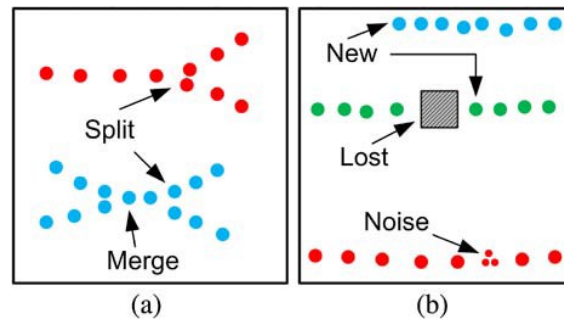


Fig. 9.7 (a) Illustration of merged and split objects.
(b) Illustration of noise, new object and lost objects



segmentation algorithm might fail resulting in an object being lost and/or new objects appearing. All these issues might occur simultaneously clouding the matter further. In Fig. 9.7 some of these phenomena are illustrated. In Fig. 9.7(a) we have a situation where one object is occluded by another object when entering the image's field of view. This continues until they split into two objects. We also see a situation where two objects merge into one and later split into two objects again. In Fig. 9.7(b) we first see a situation where a new object is detected in the middle of the scene and below a situation where an object disappears behind a static object in the scene before reappearing again. Last we see a situation where the detection of an object is incorrect resulting in the predicted object being lost and three new objects appearing.

One approach for resolving these issues is to measure how many of the detected objects are within each predicted ROI. In Fig. 9.8(a) we show an example where we have predicted five objects and detected five objects.

The zeros and ones in the table in Fig. 9.8(b) indicate if a detected object is within a predicted object's ROI. The numbers in the row (green) and column (red) outside the table indicate the sum of a particular row or column. Entries in the table those row and column sums are both one, have a unique match and can be assigned to each other. This will give that object 1 is assigned to trajectory B and object 2 to trajectory A. The row sum of the detected object 5 is equal to 0 meaning that this is a new object. The column sum of the predicted object E is 0 meaning that object E is lost. Next we look at non-assigned predicted objects with a column sum equal to 1 and assign these objects. In our example this will mean that detected object 4 is assigned to trajectory D. We therefore set entry $(C, 4) = 0$ in the table and can now assign object 3 to trajectory C since both its row and column sums are one. The final result for this image is shown to the right in Fig. 9.8(b). Looking at Fig. 9.8(a) it might be reasonable to assume that object 5 should be assigned to trajectory E. We can handle such situations by increasing the size of the ROI, but this is a dangerous path to fol-

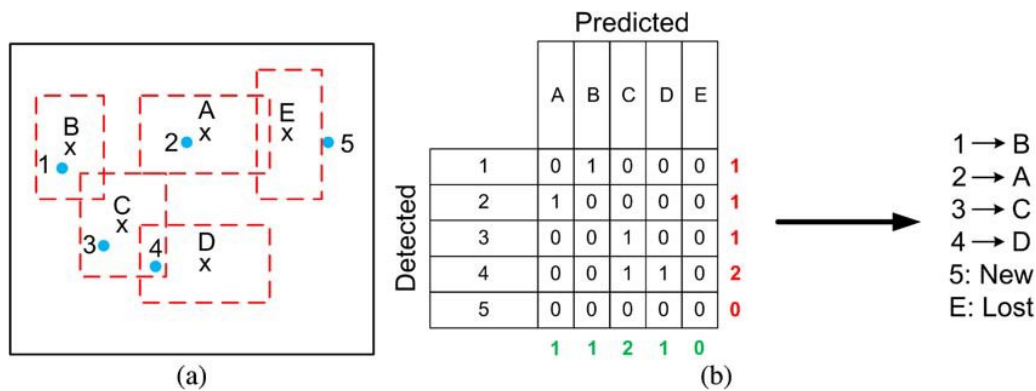


Fig. 9.8 (a) The *blue dots* are the detected objects and X illustrates the predicted objects. The *dashed red boxes* indicate the predicted ROIs. (b) A table indicating which detected objects that match which predicted objects

low since this will in general increase the number of ambiguities. A better approach is to delay the decision about whether an object is new or lost for some time.

If a lost object is present in an image, the trajectory is updated with the predicted value instead of the missing detection. The more times this is done, the more uncertain this trajectory becomes and hence the size of the ROI should be increased accordingly. Moreover, if no detections have been associated to a trajectory for some time, it should be concluded that the object is lost and its trajectory terminated.

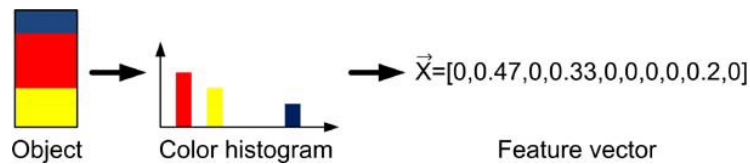
For a new object to be accepted as a truly new object the following can be done. The first time a new object is detected a temporary trajectory is defined and the object is being tracked. When it has been successfully tracked for a certain amount of time it can be concluded that this is indeed a new object and the trajectory is no longer temporary. If no detected object is associated to the temporary trajectory for some time, the temporary trajectory is terminated.

9.3.1 Good Features to Track

Instead of only focusing on the position when tracking objects we can also include the features we are using to classify the different objects. This basically means we are combining the matching problem describe above with the feature classification problem discussed in Chap. 7. In practice we base the matching on the approach from Sect. 7.3 and simply add the x- and y-positions of the object as two additional features. The binary table in Fig. 9.8(b) is then replaced by a table where each entry indicates the distance from a predicted object and to a detected object. The uncertainties related to the predicted and detected objects could/should be incorporated as weights as discussed in Sect. 7.4. To binarize this new table each entry is thresholded and we can therefore apply the same matching mechanisms as described above.

When tracking objects we can of course use any of the features described in Chap. 7. But when it comes to tracking multiple objects we usually require more details features. Below we describe two approaches namely color-based and texture-based.

Fig. 9.9 A color histogram with ten bins and how an object will be represented using the color histogram bins as features



The average color of an object can be a strong feature as it is relatively independent on how the shape and size of an object changes. Also, if a color space, where the intensity and chromaticity are separated, is used, the color feature is relatively robust to changes in the lighting. Sometimes an object contains multiple colors and the average may not be the best way to represent such an object. Instead a color histogram can be used. No matter which color space is used the different color components are concatenated and hence results in one histogram. Each histogram bin is normalized so the sum of all bins is equal to one. This makes the color histogram invariant to the scale of the object. To reduce the number of features, the resolution of the histogram bins is usually coarse. An example of a color histogram with ten bin, i.e., ten features, can be seen in Fig. 9.9.

While a color histogram is a better representation than the average color, it does not contain any information about spatial distribution of the different colors. Another approach is therefore to divide the object into a number of regions (usually horizontal dividers) and then represent each region by its average color (or color histogram). This approach is obvious sensitive to object rotation and care should therefore be taking before applying it.

As mentioned above the framerate will often be high compared to the movement of the object and it can therefore be assumed that the object does not change significantly from image to image. Inspired by this notion we can simply represent the object by its pixels and try to refind the object in the next image using template matching, see Sect. 5.2.1. For this to work the object (or a part of it) needs to be represented by a rectangle, but more importantly it is assumed that this rectangle is unique compared to the surroundings. Uniqueness here means the rectangle contains texture—the more the better—which is not repeated in the background. The level of texture can be investigated by looking at the amount of edges in the rectangle. If many strong edges are present with different orientation, then there is a high likelihood that the rectangle is unique and can be refound in the next image. One concrete way of measuring this is to correlate the rectangle with the Sobel kernels from Sect. 5.2.2. This will produce two edge images. For each edge image the absolute value of each edge pixel is found and all these values are summed, and checked if the sum is above a threshold value. We do the same for the other edge image and if both sums are above the threshold value the rectangle is concluded to contain a high level of texture, hence be a good template to track.

No matter which of these features are applied in tracking, care should be taking when combining them with the position and/or other features in order to ensure the different features are scaled properly, see Sect. 7.3. Another important issue is

that the model for a particular object is very likely to change over time and should therefore be updated from image to image. The simple solution is to replace the model with the detection, but this is dangerous since the detection could be incorrect. A gradual update scheme, like in Eq. 9.3, is therefore suggested.

9.4 Further Information

An excellent way of implementing the predict-match-update framework is through the *Kalman filter* [19]. It does not cover the detection and matching blocks, but it has built-in mechanisms for updating the state based on the detections, the predictions and the related uncertainties. When it comes to tracking noisy detections, a branch of methods exist, which do not only predict where the objects are most likely to be, but also predict a number of likely hypotheses and maintain those over time. Such methods are known as Particle Filters, the Condensation algorithm, Sequential Monte Carlo filtering, or Multiple Hypothesis filters. One place to start a journey into such methods is [11].

Color features can be improved by also including information about position. One such method is the color correlogram [20]. But when it comes to more advanced tracking, texture is often preferred over color. A good tracking framework based on texture is the KLT-tracker [16]. It finds candidate rectangles containing a high level of texture and tracks these rectangles over time. The rectangles are small and a number of these should therefore be used to track a large object. The tracker detects when the texture of a particular rectangle has changed too much compared to when it was initiated and the tracker then reinitializes a new rectangle to be tracked.

If the texture changes too much between two images, template matching-based methods will not suffice and more advanced methods are required. A good example is the SIFT algorithm [13]. It represents the pixels in a rectangle by their gradient information. This is done in a clever way making the representation invariant to rotation and scale. In Fig. 9.10 an example is shown where the object is standing still, but the camera is moving. This is equivalent to when the camera is fixed and the object is moving. The SIFT algorithm is here used to find and track 100 points between two images. Note that such approaches often refer to the process of relocating features as *finding the correspondence* rather than tracking.

9.5 Exercises

Exercise 1: Explain the following concepts: state vector, trajectory, prediction, motion model, tracking, tracking-by-detection, data association, predict-match-update framework.