

Tracking and Motion

The Basics of Tracking

When we are dealing with a video source, as opposed to individual still images, we often have a particular object or objects that we would like to follow through the visual field. In the previous chapter, we saw how to isolate a particular shape, such as a person or an automobile, on a frame-by-frame basis. Now what we'd like to do is understand the motion of this object, a task that has two main components: identification and modeling.

Identification amounts to finding the object of interest from one frame in a subsequent frame of the video stream. Techniques such as moments or color histograms from previous chapters will help us identify the object we seek. Tracking things that we have not yet identified is a related problem. Tracking unidentified objects is important when we wish to determine what is interesting based on its motion—or when an object's motion is precisely what makes it interesting. Techniques for tracking unidentified objects typically involve tracking visually significant key points (more soon on what constitutes “significance”), rather than extended objects. OpenCV provides two methods for achieving this: the Lucas-Kanade* [Lucas81] and Horn-Schunck [Horn81] techniques, which represent what are often referred to as *sparse* or *dense* optical flow respectively.

The second component, modeling, helps us address the fact that these techniques are really just providing us with noisy measurement of the object's actual position. Many powerful mathematical techniques have been developed for estimating the trajectory of an object measured in such a noisy manner. These methods are applicable to two- or three-dimensional models of objects and their locations.

Corner Finding

There are many kinds of local features that one can track. It is worth taking a moment to consider what exactly constitutes such a feature. Obviously, if we pick a point on a large blank wall then it won't be easy to find that same point in the next frame of a video.

* Oddly enough, the definitive description of Lucas-Kanade optical flow in a pyramid framework implemented in OpenCV is an unpublished paper by Bouguet [Bouguet04].

If all points on the wall are identical or even very similar, then we won't have much luck tracking that point in subsequent frames. On the other hand, if we choose a point that is unique then we have a pretty good chance of finding that point again. In practice, the point or feature we select should be unique, or nearly unique, and should be parameterizable in such a way that it can be compared to other points in another image. See Figure 10-1.

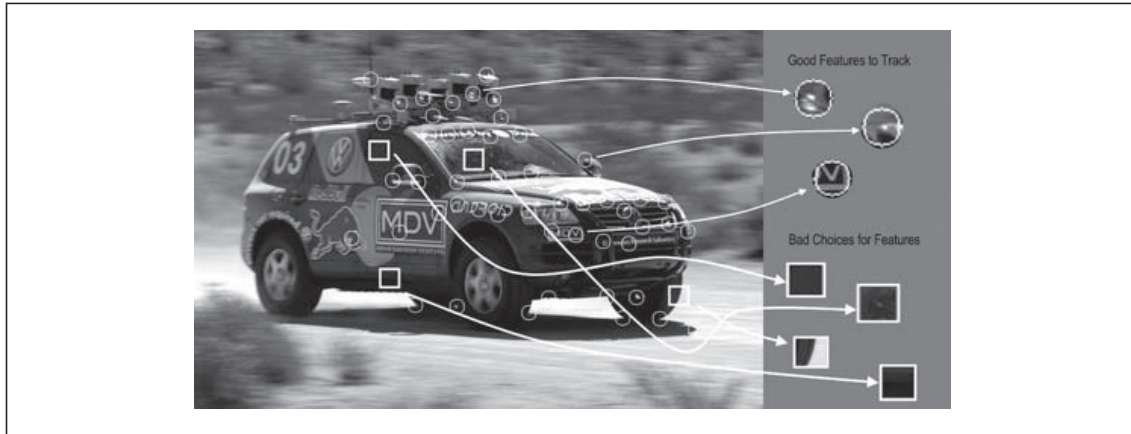


Figure 10-1. The points in circles are good points to track, whereas those in boxes—even sharply defined edges—are poor choices

Returning to our intuition from the large blank wall, we might be tempted to look for points that have some significant change in them—for example, a strong derivative. It turns out that this is not enough, but it's a start. A point to which a strong derivative is associated may be on an edge of some kind, but it could look like all of the other points along the same edge (see the aperture problem diagrammed in Figure 10-8 and discussed in the section titled “Lucas-Kanade Technique”).

However, if strong derivatives are observed in two orthogonal directions then we can hope that this point is more likely to be unique. For this reason, many trackable features are called *corners*. Intuitively, corners—not edges—are the points that contain enough information to be picked out from one frame to the next.

The most commonly used definition of a corner was provided by Harris [Harris88]. This definition relies on the matrix of the second-order derivatives ($\partial^2 x$, $\partial^2 y$, $\partial x \partial y$) of the image intensities. We can think of the second-order derivatives of images, taken at all points in the image, as forming new “second-derivative images” or, when combined together, a new *Hessian* image. This terminology comes from the Hessian matrix around a point, which is defined in two dimensions by:

$$H(p) = \begin{bmatrix} \frac{\partial^2 I}{\partial x^2} & \frac{\partial^2 I}{\partial x \partial y} \\ \frac{\partial^2 I}{\partial y \partial x} & \frac{\partial^2 I}{\partial y^2} \end{bmatrix}_p$$

For the Harris corner, we consider the *autocorrelation matrix* of the second derivative images over a small window around each point. Such a matrix is defined as follows:

$$M(x, y) = \begin{bmatrix} \sum_{-K \leq i, j \leq K} w_{i,j} I_x^2(x+i, y+j) & \sum_{-K \leq i, j \leq K} w_{i,j} I_x(x+i, y+j) I_y(x+i, y+j) \\ \sum_{-K \leq i, j \leq K} w_{i,j} I_x(x+i, y+j) I_y(x+i, y+j) & \sum_{-K \leq i, j \leq K} w_{i,j} I_y^2(x+i, y+j) \end{bmatrix}$$

(Here $w_{i,j}$ is a weighting term that can be uniform but is often used to create a circular window or Gaussian weighting.) Corners, by Harris's definition, are places in the image where the autocorrelation matrix of the second derivatives has two large eigenvalues. In essence this means that there is texture (or edges) going in at least two separate directions centered around such a point, just as real corners have at least two edges meeting in a point. Second derivatives are useful because they do not respond to uniform gradients.* This definition has the further advantage that, when we consider only the eigenvalues of the autocorrelation matrix, we are considering quantities that are invariant also to rotation, which is important because objects that we are tracking might rotate as well as move. Observe also that these two eigenvalues do more than determine if a point is a good feature to track; they also provide an identifying signature for the point.

Harris's original definition involved taking the determinant of $H(p)$, subtracting the trace of $H(p)$ (with some weighting coefficient), and then comparing this difference to a predetermined threshold. It was later found by Shi and Tomasi [Shi94] that good corners resulted as long as the smaller of the two eigenvalues was greater than a minimum threshold. Shi and Tomasi's method was not only sufficient but in many cases gave more satisfactory results than Harris's method.

The `cvGoodFeaturesToTrack()` routine implements the Shi and Tomasi definition. This function conveniently computes the second derivatives (using the Sobel operators) that are needed and from those computes the needed eigenvalues. It then returns a list of the points that meet our definition of being good for tracking.

```
void cvGoodFeaturesToTrack(
    const CvArr*    image,
    CvArr*          eigImage,
    CvArr*          tempImage,
    CvPoint2D32f*   corners,
    int*            corner_count,
    double          quality_level,
    double          min_distance,
    const CvArr*    mask          = NULL,
    int             block_size    = 3,
    int             use_harris    = 0,
    double          k             = 0.4
);
```

* A gradient is derived from first derivatives. If first derivatives are uniform (constant), then second derivatives are 0.

In this case, the input image should be an 8-bit or 32-bit (i.e., `IPL_DEPTH_8U` or `IPL_DEPTH_32F`) single-channel image. The next two arguments are single-channel 32-bit images of the same size. Both `tempImage` and `eigImage` are used as scratch by the algorithm, but the resulting contents of `eigImage` are meaningful. In particular, each entry there contains the minimal eigenvalue for the corresponding point in the input image. Here `corners` is an array of 32-bit points (`CvPoint2D32f`) that contain the result points after the algorithm has run; you must allocate this array before calling `cvGoodFeaturesToTrack()`. Naturally, since you allocated that array, you only allocated a finite amount of memory. The `corner_count` indicates the maximum number of points for which there is space to return. After the routine exits, `corner_count` is overwritten by the number of points that were actually found. The parameter `quality_level` indicates the minimal acceptable lower eigenvalue for a point to be included as a corner. The actual minimal eigenvalue used for the cutoff is the product of the `quality_level` and the largest lower eigenvalue observed in the image. Hence, the `quality_level` should not exceed 1 (a typical value might be 0.10 or 0.01). Once these candidates are selected, a further culling is applied so that multiple points within a small region need not be included in the response. In particular, the `min_distance` guarantees that no two returned points are within the indicated number of pixels.

The optional `mask` is the usual image, interpreted as Boolean values, indicating which points should and which points should not be considered as possible corners. If set to `NULL`, no mask is used. The `block_size` is the region around a given pixel that is considered when computing the autocorrelation matrix of derivatives. It turns out that it is better to sum these derivatives over a small window than to compute their value at only a single point (i.e., at a `block_size` of 1). If `use_harris` is nonzero, then the Harris corner definition is used rather than the Shi-Tomasi definition. If you set `use_harris` to a nonzero value, then the value `k` is the weighting coefficient used to set the relative weight given to the trace of the autocorrelation matrix Hessian compared to the determinant of the same matrix.

Once you have called `cvGoodFeaturesToTrack()`, the result is an array of pixel locations that you hope to find in another similar image. For our current context, we are interested in looking for these features in subsequent frames of video, but there are many other applications as well. A similar technique can be used when attempting to relate multiple images taken from slightly different viewpoints. We will re-encounter this issue when we discuss stereo vision in later chapters.

Subpixel Corners

If you are processing images for the purpose of extracting geometric measurements, as opposed to extracting features for recognition, then you will normally need more resolution than the simple pixel values supplied by `cvGoodFeaturesToTrack()`. Another way of saying this is that such pixels come with integer coordinates whereas we sometimes require real-valued coordinates—for example, pixel (8.25, 117.16).

One might imagine needing to look for a sharp peak in image values, only to be frustrated by the fact that the peak's location will almost never be in the exact center of a

camera pixel element. To overcome this, you might fit a curve (say, a parabola) to the image values and then use a little math to find where the peak occurred between the pixels. Subpixel detection techniques are all about tricks like this (for a review and newer techniques, see Lucchese [Lucchese02] and Chen [Chen05]). Common uses of image measurements are tracking for three-dimensional reconstruction, calibrating a camera, warping partially overlapping views of a scene to stitch them together in the most natural way, and finding an external signal such as precise location of a building in a satellite image.

Subpixel corner locations are a common measurement used in camera calibration or when tracking to reconstruct the camera's path or the three-dimensional structure of a tracked object. Now that we know how to find corner locations on the integer grid of pixels, here's the trick for refining those locations to subpixel accuracy: We use the mathematical fact that the dot product between a vector and an orthogonal vector is 0; this situation occurs at corner locations, as shown in Figure 10-2.

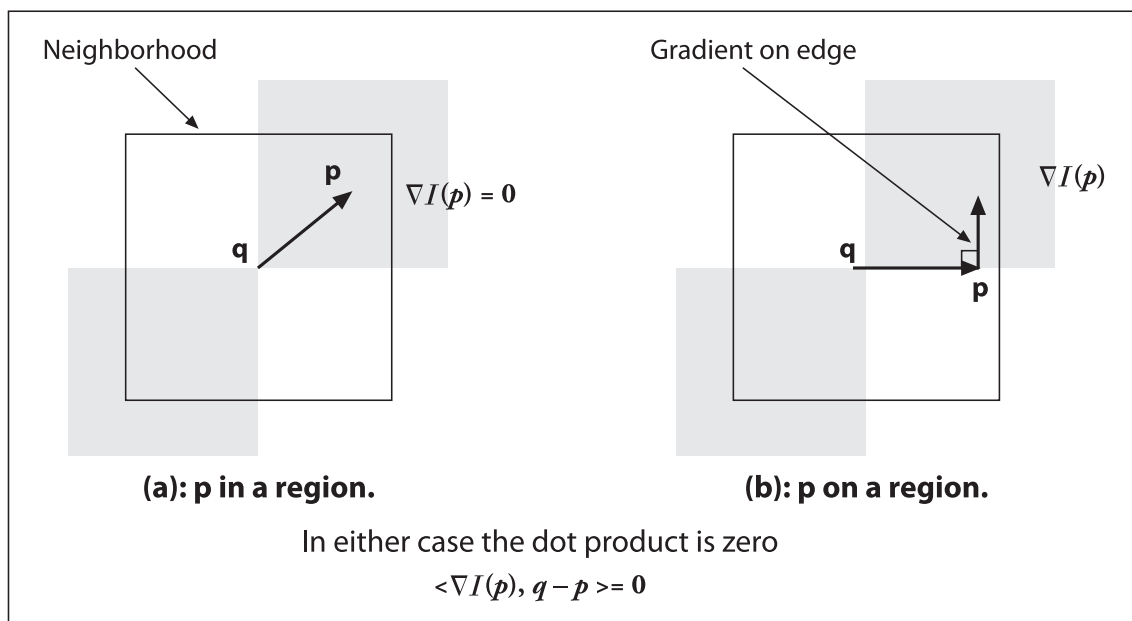


Figure 10-2. Finding corners to subpixel accuracy: (a) the image area around the point p is uniform and so its gradient is 0; (b) the gradient at the edge is orthogonal to the vector $q-p$ along the edge; in either case, the dot product between the gradient at p and the vector $q-p$ is 0 (see text)

In the figure, we assume a starting corner location q that is near the actual subpixel corner location. We examine vectors starting at point q and ending at p . When p is in a nearby uniform or “flat” region, the gradient there is 0. On the other hand, if the vector $q-p$ aligns with an edge then the gradient at p on that edge is orthogonal to the vector $q-p$. In either case, the dot product between the gradient at p and the vector $q-p$ is 0. We can assemble many such pairs of the gradient at a nearby point p and the associated vector $q-p$, set their dot product to 0, and solve this assemblage as a system of equations; the solution will yield a more accurate subpixel location for q , the exact location of the corner.

The function that does subpixel corner finding is `cvFindCornerSubPix()`:

```
void cvFindCornerSubPix(  
    const CvArr*    image,  
    CvPoint2D32f*   corners,  
    int             count,  
    CvSize          win,  
    CvSize          zero_zone,  
    CvTermCriteria  criteria  
);
```

The input `image` is a single-channel, 8-bit, grayscale image. The `corners` structure contains integer pixel locations, such as those obtained from routines like `cvGoodFeaturesToTrack()`, which are taken as the initial guesses for the corner locations; `count` holds how many points there are to compute.

The actual computation of the subpixel location uses a system of dot-product expressions that all equal 0 (see Figure 10-2), where each equation arises from considering a single pixel in the region around p . The parameter `win` specifies the size of window from which these equations will be generated. This window is centered on the original integer corner location and extends outward in each direction by the number of pixels specified in `win` (e.g., if `win.width = 4` then the search area is actually $4 + 1 + 4 = 9$ pixels wide). These equations form a linear system that can be solved by the inversion of a single autocorrelation matrix (not related to the autocorrelation matrix encountered in our previous discussion of Harris corners). In practice, this matrix is not always invertible owing to small eigenvalues arising from the pixels very close to p . To protect against this, it is common to simply reject from consideration those pixels in the immediate neighborhood of p . The parameter `zero_zone` defines a window (analogously to `win`, but always with a smaller extent) that will *not* be considered in the system of constraining equations and thus the autocorrelation matrix. If no such zero zone is desired then this parameter should be set to `cvSize(-1,-1)`.

Once a new location is found for q , the algorithm will iterate using that value as a starting point and will continue until the user-specified termination criterion is reached. Recall that this criterion can be of type `CV_TERMCRIT_ITER` or of type `CV_TERMCRIT_EPS` (or both) and is usually constructed with the `cvTermCriteria()` function. Using `CV_TERMCRIT_EPS` will effectively indicate the accuracy you require of the subpixel values. Thus, if you specify 0.10 then you are asking for subpixel accuracy down to one tenth of a pixel.

Invariant Features

Since the time of Harris's original paper and the subsequent work by Shi and Tomasi, a great many other types of corners and related local features have been proposed. One widely used type is the SIFT ("scale-invariant feature transform") feature [Lowe04]. Such features are, as their name suggests, scale-invariant. Because SIFT detects the dominant gradient orientation at its location and records its local gradient histogram results with respect to this orientation, SIFT is also rotationally invariant. As a result, SIFT features are relatively well behaved under small affine transformations. Although the SIFT

algorithm is not yet implemented as part of the OpenCV library (but see Chapter 14), it is possible to create such an implementation using OpenCV primitives. We will not spend more time on this topic, but it is worth keeping in mind that, given the OpenCV functions we've already discussed, it is possible (albeit less convenient) to create most of the features reported in the computer vision literature (see Chapter 14 for a feature tool kit in development).

Optical Flow

As already mentioned, you may often want to assess motion between two frames (or a sequence of frames) without any other prior knowledge about the content of those frames. Typically, the motion itself is what indicates that something interesting is going on. Optical flow is illustrated in Figure 10-3.

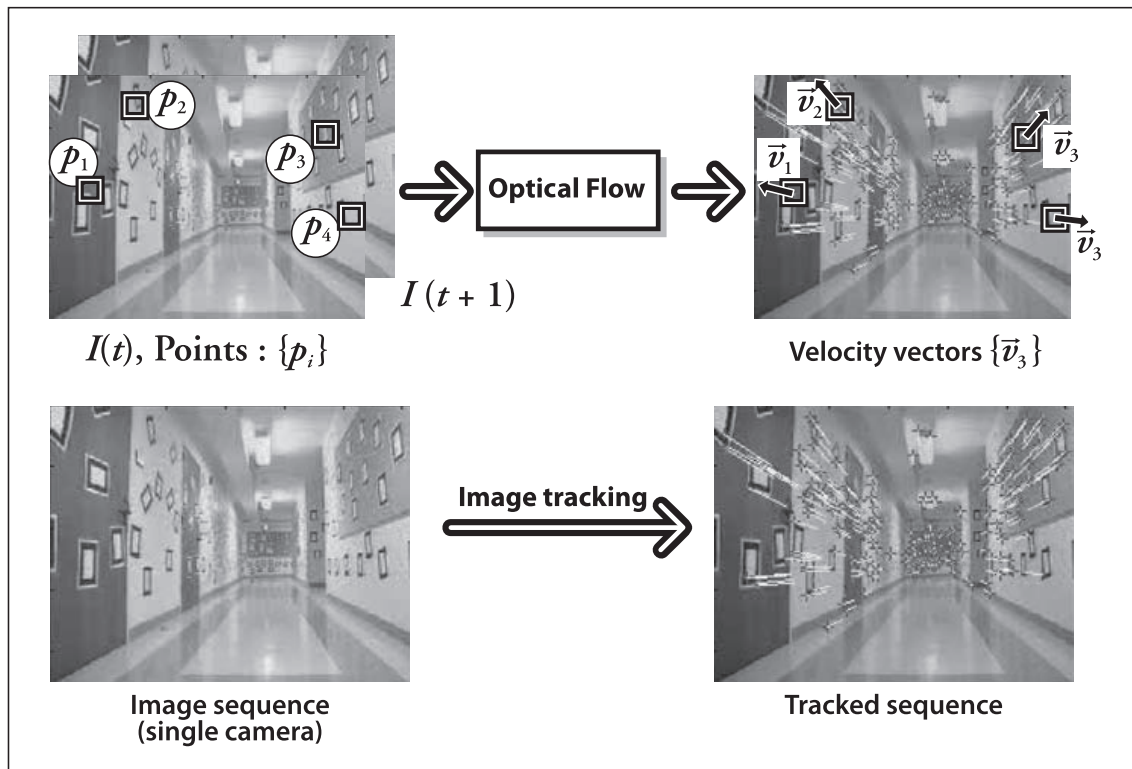


Figure 10-3. Optical flow: target features (upper left) are tracked over time and their movement is converted into velocity vectors (upper right); lower panels show a single image of the hallway (left) and flow vectors (right) as the camera moves down the hall (original images courtesy of Jean-Yves Bouguet)

We can associate some kind of velocity with each pixel in the frame or, equivalently, some displacement that represents the distance a pixel has moved between the previous frame and the current frame. Such a construction is usually referred to as a *dense optical flow*, which associates a velocity with every pixel in an image. The *Horn-Schunck method* [Horn81] attempts to compute just such a velocity field. One seemingly straightforward method—simply attempting to match windows around each pixel from one frame to

the next—is also implemented in OpenCV; this is known as *block matching*. Both of these routines will be discussed in the “Dense Tracking Techniques” section.

In practice, calculating dense optical flow is not easy. Consider the motion of a white sheet of paper. Many of the white pixels in the previous frame will simply remain white in the next. Only the edges may change, and even then only those perpendicular to the direction of motion. The result is that dense methods must have some method of interpolating between points that are more easily tracked so as to solve for those points that are more ambiguous. These difficulties manifest themselves most clearly in the high computational costs of dense optical flow.

This leads us to the alternative option, *sparse optical flow*. Algorithms of this nature rely on some means of specifying beforehand the subset of points that are to be tracked. If these points have certain desirable properties, such as the “corners” discussed earlier, then the tracking will be relatively robust and reliable. We know that OpenCV can help us by providing routines for identifying the best features to track. For many practical applications, the computational cost of sparse tracking is so much less than dense tracking that the latter is relegated to only academic interest.*

The next few sections present some different methods of tracking. We begin by considering the most popular sparse tracking technique, *Lucas-Kanade* (LK) optical flow; this method also has an implementation that works with image pyramids, allowing us to track faster motions. We’ll then move on to two dense techniques, the Horn-Schunck method and the block matching method.

Lucas-Kanade Method

The Lucas-Kanade (LK) algorithm [Lucas81], as originally proposed in 1981, was an attempt to produce dense results. Yet because the method is easily applied to a subset of the points in the input image, it has become an important sparse technique. The LK algorithm can be applied in a sparse context because it relies only on local information that is derived from some small window surrounding each of the points of interest. This is in contrast to the intrinsically global nature of the Horn and Schunck algorithm (more on this shortly). The disadvantage of using small local windows in Lucas-Kanade is that large motions can move points outside of the local window and thus become impossible for the algorithm to find. This problem led to development of the “pyramidal” LK algorithm, which tracks starting from highest level of an image pyramid (lowest detail) and working down to lower levels (finer detail). Tracking over image pyramids allows large motions to be caught by local windows.

Because this is an important and effective technique, we shall go into some mathematical detail; readers who prefer to forgo such details can skip to the function description and code. However, it is recommended that you at least scan the intervening text and

* Black and Anadan have created dense optical flow techniques [Black93; Black96] that are often used in movie production, where, for the sake of visual quality, the movie studio is willing to spend the time necessary to obtain detailed flow information. These techniques are slated for inclusion in later versions of OpenCV (see Chapter 14).

figures, which describe the assumptions behind Lucas-Kanade optical flow, so that you'll have some intuition about what to do if tracking isn't working well.

How Lucas-Kanade works

The basic idea of the LK algorithm rests on three assumptions.

1. *Brightness constancy*. A pixel from the image of an object in the scene does not change in appearance as it (possibly) moves from frame to frame. For grayscale images (LK can also be done in color), this means we assume that the brightness of a pixel does not change as it is tracked from frame to frame.
2. *Temporal persistence or "small movements"*. The image motion of a surface patch changes slowly in time. In practice, this means the temporal increments are fast enough relative to the scale of motion in the image that the object does not move much from frame to frame.
3. *Spatial coherence*. Neighboring points in a scene belong to the same surface, have similar motion, and project to nearby points on the image plane.

We now look at how these assumptions, which are illustrated in Figure 10-4, lead us to an effective tracking algorithm. The first requirement, brightness constancy, is just the requirement that pixels in one tracked patch look the same over time:

$$f(x, t) \equiv I(x(t), t) = I(x(t+dt), t+dt)$$

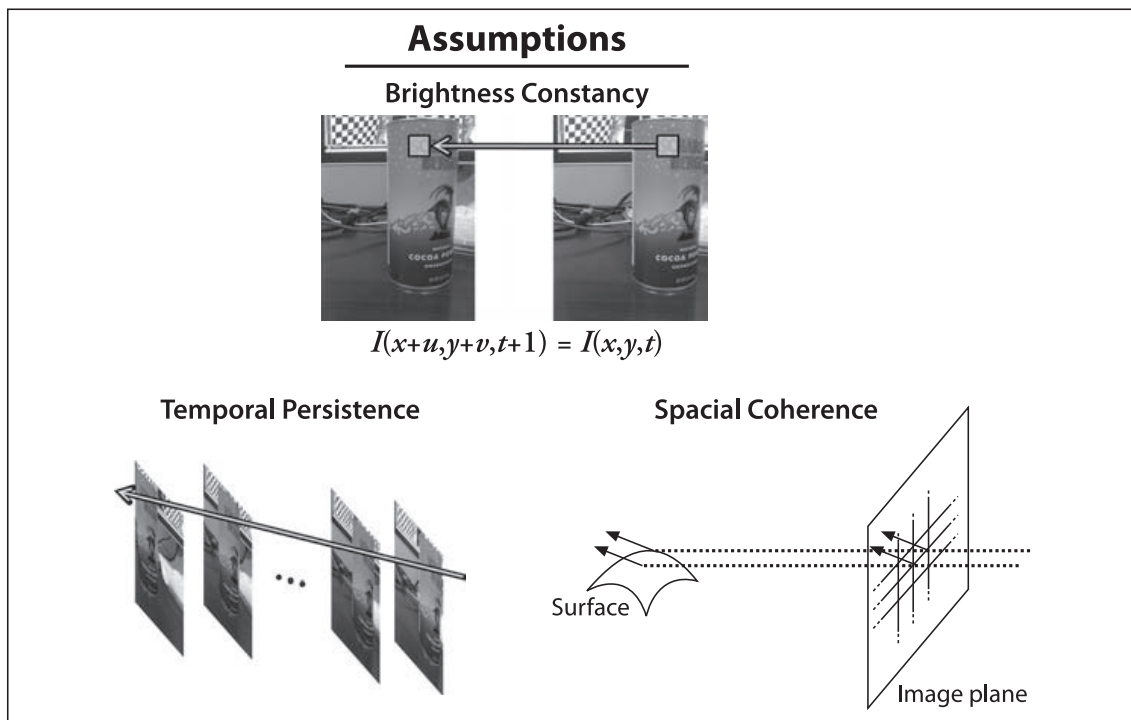


Figure 10-4. Assumptions behind Lucas-Kanade optical flow: for a patch being tracked on an object in a scene, the patch's brightness doesn't change (top); motion is slow relative to the frame rate (lower left); and neighboring points stay neighbors (lower right) (component images courtesy of Michael Black [Black82])

That's simple enough, and it means that our tracked pixel intensity exhibits no change over time:

$$\frac{\partial f(x)}{\partial t} = 0$$

The second assumption, temporal persistence, essentially means that motions are small from frame to frame. In other words, we can view this change as approximating a derivative of the intensity with respect to time (i.e., we assert that the change between one frame and the next in a sequence is *differentially small*). To understand the implications of this assumption, first consider the case of a single spatial dimension.

In this case we can start with our brightness consistency equation, substitute the definition of the brightness $f(x, t)$ while taking into account the implicit dependence of x on t , $I(x(t), t)$, and then apply the chain rule for partial differentiation. This yields:

$$\underbrace{\frac{\partial I}{\partial x}}_{I_x} \underbrace{\left(\frac{\partial x}{\partial t} \right)}_{\mathbf{v}} + \underbrace{\frac{\partial I}{\partial t}}_{I_t} = 0$$

where I_x is the spatial derivative across the first image, I_t is the derivative between images over time, and \mathbf{v} is the velocity we are looking for. We thus arrive at the simple equation for optical flow velocity in the simple one-dimensional case:

$$\mathbf{v} = -\frac{I_t}{I_x}$$

Let's now try to develop some intuition for the one-dimensional tracking problem. Consider Figure 10-5, which shows an “edge”—consisting of a high value on the left and a low value on the right—that is moving to the right along the x -axis. Our goal is to identify the velocity \mathbf{v} at which the edge is moving, as plotted in the upper part of Figure 10-5. In the lower part of the figure we can see that our measurement of this velocity is just “rise over run,” where the rise is over time and the run is the slope (spatial derivative). The negative sign corrects for the slope of x .

Figure 10-5 reveals another aspect to our optical flow formulation: our assumptions are probably not quite true. That is, image brightness is not really stable; and our time steps (which are set by the camera) are often not as fast relative to the motion as we'd like. Thus, our solution for the velocity is not exact. However, if we are “close enough” then we can iterate to a solution. Iteration is shown in Figure 10-6, where we use our first (inaccurate) estimate of velocity as the starting point for our next iteration and then repeat. Note that we can keep the same spatial derivative in x as computed on the first frame because of the brightness constancy assumption—pixels moving in x do not change. This reuse of the spatial derivative already calculated yields significant computational savings. The time derivative must still be recomputed each iteration and each frame, but

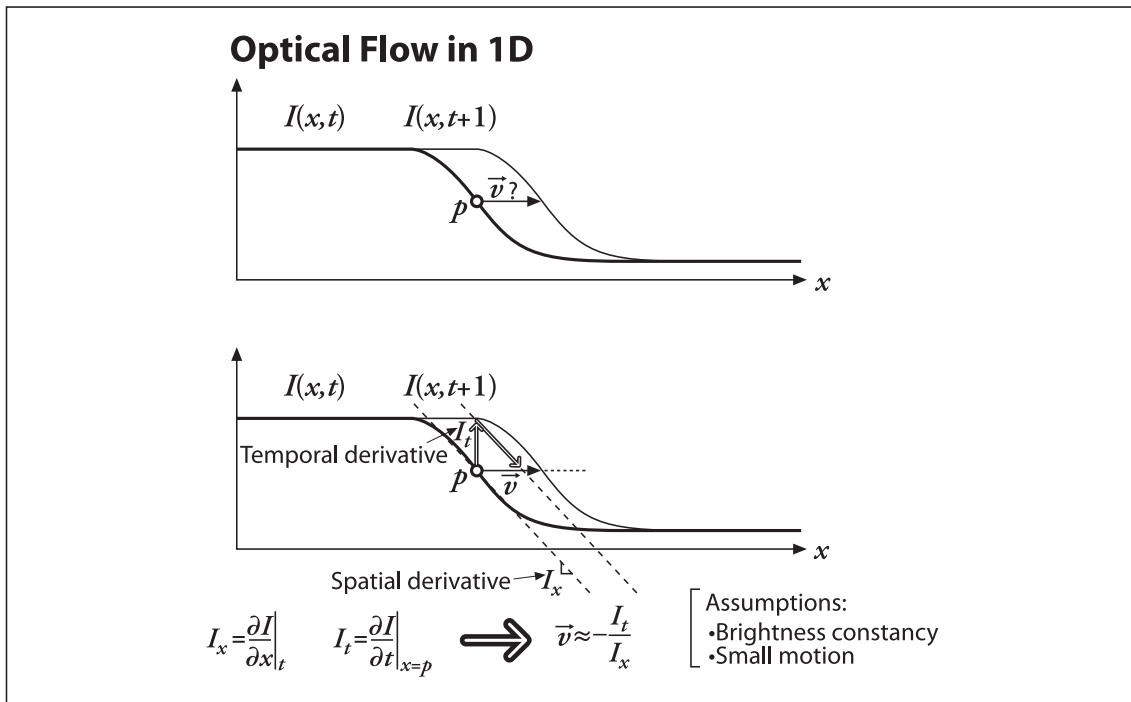


Figure 10-5. Lucas-Kanade optical flow in one dimension: we can estimate the velocity of the moving edge (upper panel) by measuring the ratio of the derivative of the intensity over time divided by the derivative of the intensity over space

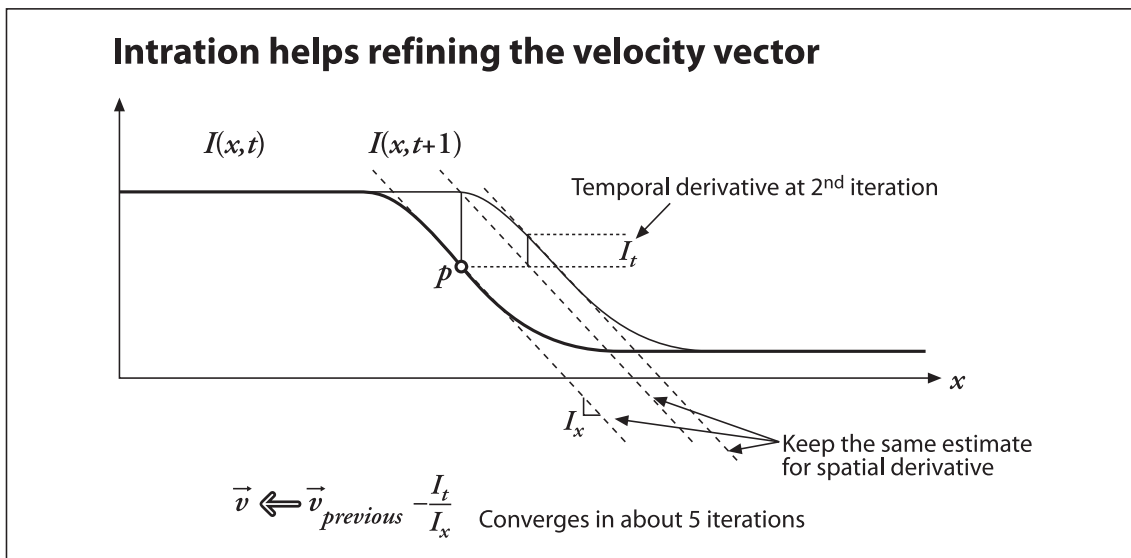


Figure 10-6. Iterating to refine the optical flow solution (Newton's method): using the same two images and the same spatial derivative (slope) we solve again for the time derivative; convergence to a stable solution usually occurs within a few iterations

if we are close enough to start with then these iterations will converge to near exactitude within about five iterations. This is known as *Newton's method*. If our first estimate was not close enough, then Newton's method will actually diverge.

Now that we've seen the one-dimensional solution, let's generalize it to images in two dimensions. At first glance, this seems simple: just add in the y coordinate. Slightly

changing notation, we'll call the y component of velocity v and the x component of velocity u ; then we have:

$$I_x u + I_y v + I_t = 0$$

Unfortunately, for this single equation there are two unknowns for any given pixel. This means that measurements at the single-pixel level are underconstrained and cannot be used to obtain a unique solution for the two-dimensional motion at that point. Instead, we can only solve for the motion component that is perpendicular or “normal” to the line described by our flow equation. Figure 10-7 presents the mathematical and geometric details.

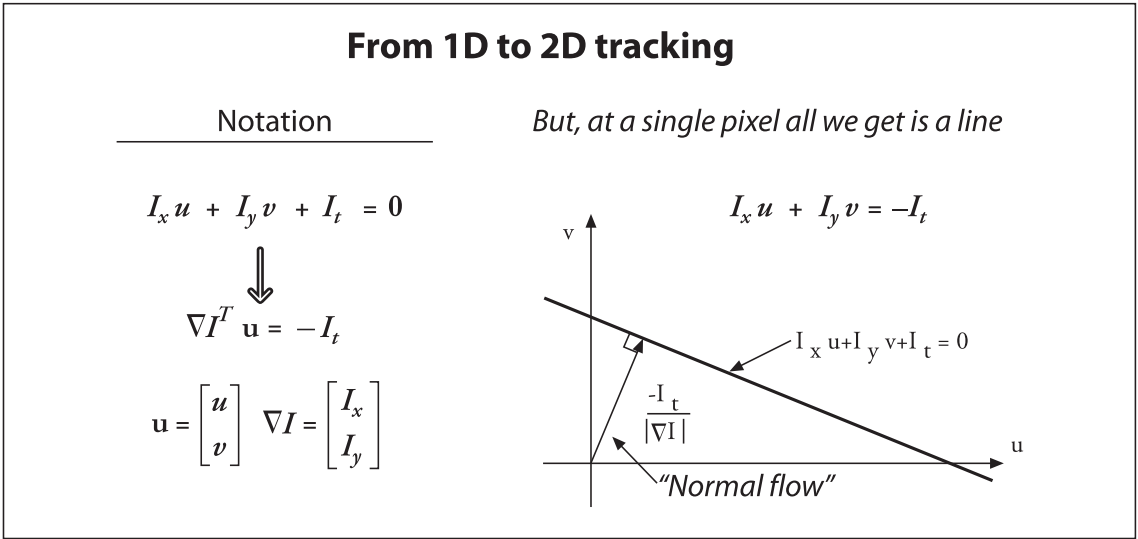


Figure 10-7. Two-dimensional optical flow at a single pixel: optical flow at one pixel is underdetermined and so can yield at most motion, which is perpendicular (“normal”) to the line described by the flow equation (figure courtesy of Michael Black)

Normal optical flow results from the *aperture problem*, which arises when you have a small aperture or window in which to measure motion. When motion is detected with a small aperture, you often see only an edge, not a corner. But an edge alone is insufficient to determine exactly how (i.e., in what direction) the entire object is moving; see Figure 10-8.

So then how do we get around this problem that, at one pixel, we cannot resolve the full motion? We turn to the last optical flow assumption for help. If a local patch of pixels moves coherently, then we can easily solve for the motion of the central pixel by using the surrounding pixels to set up a system of equations. For example, if we use a 5-by-5* window of brightness values (you can simply triple this for color-based optical flow) around the current pixel to compute its motion, we can then set up 25 equations as follows.

* Of course, the window could be 3-by-3, 7-by-7, or anything you choose. If the window is too large then you will end up violating the coherent motion assumption and will not be able to track well. If the window is too small, you will encounter the aperture problem again.

$$\underbrace{\begin{bmatrix} I_x(p_1) & I_y(p_1) \\ I_x(p_2) & I_y(p_2) \\ \vdots & \vdots \\ I_x(p_{25}) & I_y(p_{25}) \end{bmatrix}}_{\substack{A \\ 25 \times 2}} \underbrace{\begin{bmatrix} u \\ v \end{bmatrix}}_{\substack{d \\ 2 \times 1}} = - \underbrace{\begin{bmatrix} I_t(p_1) \\ I_t(p_2) \\ \vdots \\ I_t(p_{25}) \end{bmatrix}}_{\substack{b \\ 25 \times 1}}$$

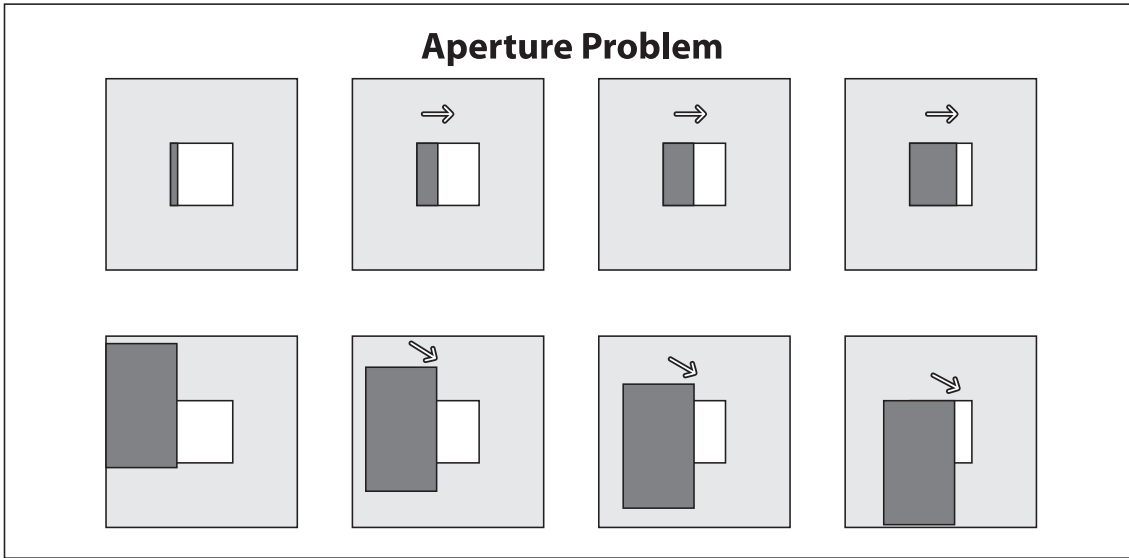


Figure 10-8. Aperture problem: through the aperture window (upper row) we see an edge moving to the right but cannot detect the downward part of the motion (lower row)

We now have an overconstrained system for which we can solve provided it contains more than just an edge in that 5-by-5 window. To solve for this system, we set up a least-squares minimization of the equation, whereby $\min \|Ad - b\|^2$ is solved in standard form as:

$$\underbrace{(A^T A)}_{2 \times 2} \underbrace{d}_{2 \times 1} = \underbrace{A^T b}_{2 \times 2}$$

From this relation we obtain our u and v motion components. Writing this out in more detail yields:

$$\underbrace{\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix}}_{A^T A} \begin{bmatrix} u \\ v \end{bmatrix} = - \underbrace{\begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}}_{A^T b}$$

The solution to this equation is then:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (A^T A)^{-1} A^T b$$

When can this be solved?—when $(A^T A)$ is invertible. And $(A^T A)$ is invertible when it has full rank (2), which occurs when it has two large eigenvectors. This will happen in image regions that include texture running in at least two directions. In this case, $(A^T A)$ will have the best properties then when the tracking window is centered over a corner region in an image. This ties us back to our earlier discussion of the Harris corner detector. In fact, those corners were “good features to track” (see our previous remarks concerning `cvGoodFeaturesToTrack()`) for precisely the reason that $(A^T A)$ had two large eigenvectors there! We’ll see shortly how all this computation is done for us by the `cvCalcOpticalFlowLK()` function.

The reader who understands the implications of our assuming small and coherent motions will now be bothered by the fact that, for most video cameras running at 30 Hz, large and noncoherent motions are commonplace. In fact, Lucas-Kanade optical flow by itself does not work very well for exactly this reason: we want a large window to catch large motions, but a large window too often breaks the coherent motion assumption! To circumvent this problem, we can track first over larger spatial scales using an image pyramid and then refine the initial motion velocity assumptions by working our way down the levels of the image pyramid until we arrive at the raw image pixels.

Hence, the recommended technique is first to solve for optical flow at the top layer and then to use the resulting motion estimates as the starting point for the next layer down. We continue going down the pyramid in this manner until we reach the lowest level. Thus we minimize the violations of our motion assumptions and so can track faster and longer motions. This more elaborate function is known as *pyramid Lucas-Kanade optical flow* and is illustrated in Figure 10-9. The OpenCV function that implements Pyramid Lucas-Kanade optical flow is `cvCalcOpticalFlowPyrLK()`, which we examine next.

Lucas-Kanade code

The routine that implements the nonpyramidal Lucas-Kanade dense optical flow algorithm is:

```
void cvCalcOpticalFlowLK(
    const CvArr* imgA,
    const CvArr* imgB,
    CvSize      winSize,
    CvArr*      velx,
    CvArr*      vely
);
```

The result arrays for this OpenCV routine are populated only by those pixels for which it is able to compute the minimum error. For the pixels for which this error (and thus the displacement) cannot be reliably computed, the associated velocity will be set to 0. In most cases, you will not want to use this routine. The following pyramid-based method is better for most situations most of the time.

Pyramid Lucas-Kanade code

We come now to OpenCV’s algorithm that computes Lucas-Kanade optical flow in a pyramid, `cvCalcOpticalFlowPyrLK()`. As we will see, this optical flow function makes use

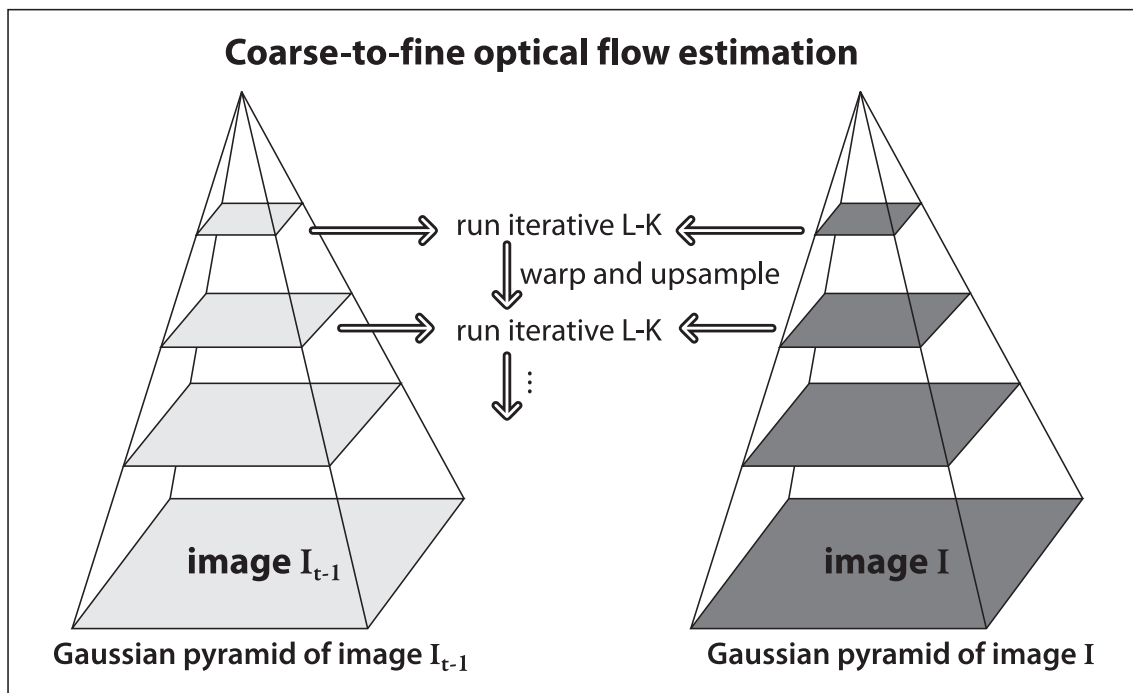


Figure 10-9. Pyramid Lucas-Kanade optical flow: running optical flow at the top of the pyramid first mitigates the problems caused by violating our assumptions of small and coherent motion; the motion estimate from the preceding level is taken as the starting point for estimating motion at the next layer down

of “good features to track” and also returns indications of how well the tracking of each point is proceeding.

```
void cvCalcOpticalFlowPyrLK(
    const CvArr*    imgA,
    const CvArr*    imgB,
    CvArr*          pyrA,
    CvArr*          pyrB,
    CvPoint2D32f*   featuresA,
    CvPoint2D32f*   featuresB,
    int             count,
    CvSize          winSize,
    int             level,
    char*           status,
    float*          track_error,
    CvTermCriteria   criteria,
    int             flags
);
```

This function has a lot of inputs, so let’s take a moment to figure out what they all do. Once we have a handle on this routine, we can move on to the problem of which points to track and how to compute them.

The first two arguments of `cvCalcOpticalFlowPyrLK()` are the initial and final images; both should be single-channel, 8-bit images. The next two arguments are buffers allocated to store the pyramid images. The size of these buffers should be at least `(img.width`

+ 8)*img.height/3 bytes,* with one such buffer for each of the two input images (pyrA and pyrB). (If these two pointers are set to NULL then the routine will allocate, use, and free the appropriate memory when called, but this is not so good for performance.) The array featuresA contains the points for which the motion is to be found, and featuresB is a similar array into which the computed new locations of the points from featuresA are to be placed; count is the number of points in the featuresA list. The window used for computing the local coherent motion is given by winSize. Because we are constructing an image pyramid, the argument level is used to set the depth of the stack of images. If level is set to 0 then the pyramids are not used. The array status is of length count; on completion of the routine, each entry in status will be either 1 (if the corresponding point was found in the second image) or 0 (if it was not). The track_error parameter is optional and can be turned off by setting it to NULL. If track_error is active then it is an array of numbers, one for each tracked point, equal to the difference between the patch around a tracked point in the first image and the patch around the location to which that point was tracked in the second image. You can use track_error to prune away points whose local appearance patch changes too much as the points move.

The next thing we need is the termination criteria. This is a structure used by many OpenCV algorithms that iterate to a solution:

```
cvTermCriteria(
    int    type,      // CV_TERMCRIT_ITER, CV_TERMCRIT_EPS, or both
    int    max_iter,
    double epsilon
);
```

Typically we use the cvTermCriteria() function to generate the structure we need. The first argument of this function is either CV_TERMCRIT_ITER or CV_TERMCRIT_EPS, which tells the algorithm that we want to terminate either after some number of iterations or when the convergence metric reaches some small value (respectively). The next two arguments set the values at which one, the other, or both of these criteria should terminate the algorithm. The reason we have both options is so we can set the type to CV_TERMCRIT_ITER | CV_TERMCRIT_EPS and thus stop when either limit is reached (this is what is done in most real code).

Finally, flags allows for some fine control of the routine's internal bookkeeping; it may be set to any or all (using bitwise OR) of the following.

CV_LKFLOW_PYR_A_READY

The image pyramid for the first frame is calculated before the call and stored in pyrA.

CV_LKFLOW_PYR_B_READY

The image pyramid for the second frame is calculated before the call and stored in pyrB.

* If you are wondering why the funny size, it's because these scratch spaces need to accommodate not just the image itself but the entire pyramid.

CV_LKFLOW_INITIAL_GUESSES

The array B already contains an initial guess for the feature's coordinates when the routine is called.

These flags are particularly useful when handling sequential video. The image pyramids are somewhat costly to compute, so recomputing them should be avoided whenever possible. The final frame for the frame pair you just computed will be the initial frame for the pair that you will compute next. If you allocated those buffers yourself (instead of asking the routine to do it for you), then the pyramids for each image will be sitting in those buffers when the routine returns. If you tell the routine that this information is already computed then it will not be recomputed. Similarly, if you computed the motion of points from the previous frame then you are in a good position to make good initial guesses for where they will be in the next frame.

So the basic plan is simple: you supply the images, list the points you want to track in featuresA, and call the routine. When the routine returns, you check the status array to see which points were successfully tracked and then check featuresB to find the new locations of those points.

This leads us back to that issue we put aside earlier: how to decide which features are good ones to track. Earlier we encountered the OpenCV routine `cvGoodFeaturesToTrack()`, which uses the method originally proposed by Shi and Tomasi to solve this problem in a reliable way. In most cases, good results are obtained by using the combination of `cvGoodFeaturesToTrack()` and `cvCalcOpticalFlowPyrLK()`. Of course, you can also use your own criteria to determine which points to track.

Let's now look at a simple example (Example 10-1) that uses both `cvGoodFeaturesToTrack()` and `cvCalcOpticalFlowPyrLK()`; see also Figure 10-10.

Example 10-1. Pyramid Lucas-Kanade optical flow code

```
// Pyramid L-K optical flow example
//
#include <cv.h>
#include <cxcore.h>
#include <highgui.h>

const int MAX_CORNERS = 500;

int main(int argc, char** argv) {

    // Initialize, load two images from the file system, and
    // allocate the images and other structures we will need for
    // results.
    //
    IplImage* imgA = cvLoadImage("image0.jpg",CV_LOAD_IMAGE_GRAYSCALE);
    IplImage* imgB = cvLoadImage("image1.jpg",CV_LOAD_IMAGE_GRAYSCALE);

    CvSize    img_sz    = cvGetSize( imgA );
    int       win_size = 10;

    IplImage* imgC = cvLoadImage(
```

Example 10-1. Pyramid Lucas-Kanade optical flow code (continued)

```
    "../Data/OpticalFlow1.jpg",
    CV_LOAD_IMAGE_UNCHANGED
);

// The first thing we need to do is get the features
// we want to track.
//
IplImage* eig_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );
IplImage* tmp_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );

int          corner_count = MAX_CORNERS;
CvPoint2D32f* cornersA    = new CvPoint2D32f[ MAX_CORNERS ];

cvGoodFeaturesToTrack(
    imgA,
    eig_image,
    tmp_image,
    cornersA,
    &corner_count,
    0.01,
    5.0,
    0,
    3,
    0,
    0.04
);

cvFindCornerSubPix(
    imgA,
    cornersA,
    corner_count,
    cvSize(win_size,win_size),
    cvSize(-1,-1),
    cvTermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS,20,0.03)
);

// Call the Lucas Kanade algorithm
//
char features_found[ MAX_CORNERS ];
float feature_errors[ MAX_CORNERS ];

CvSize pyr_sz = cvSize( imgA->width+8, imgB->height/3 );

IplImage* pyrA = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );
IplImage* pyrB = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );

CvPoint2D32f* cornersB    = new CvPoint2D32f[ MAX_CORNERS ];

cvCalcOpticalFlowPyrLK(
    imgA,
    imgB,
```

Example 10-1. Pyramid Lucas-Kanade optical flow code (continued)

```
    pyrA,
    pyrB,
    cornersA,
    cornersB,
    corner_count,
    cvSize( win_size,win_size ),
    5,
    features_found,
    feature_errors,
    cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .3 ),
    0
);

// Now make some image of what we are looking at:
//
for( int i=0; i<corner_count; i++ ) {
    if( features_found[i]==0|| feature_errors[i]>550 ) {
        printf("Error is %f/n",feature_errors[i]);
        continue;
    }
    printf("Got it/n");
    CvPoint p0 = cvPoint(
        cvRound( cornersA[i].x ),
        cvRound( cornersA[i].y )
    );
    CvPoint p1 = cvPoint(
        cvRound( cornersB[i].x ),
        cvRound( cornersB[i].y )
    );
    cvLine( imgC, p0, p1, CV_RGB(255,0,0),2 );
}

cvNamedWindow("ImageA",0);
cvNamedWindow("ImageB",0);
cvNamedWindow("LKpyr_OpticalFlow",0);

cvShowImage("ImageA",imgA);
cvShowImage("ImageB",imgB);
cvShowImage("LKpyr_OpticalFlow",imgC);

cvWaitKey(0);

return 0;
}
```

Dense Tracking Techniques

OpenCV contains two other optical flow techniques that are now seldom used. These routines are typically much slower than Lucas-Kanade; moreover, they (could, but) do not support matching within an image scale pyramid and so cannot track large motions. We will discuss them briefly in this section.



Figure 10-10. Sparse optical flow from pyramid Lucas-Kanade: the center image is one video frame after the left image; the right image illustrates the computed motion of the “good features to track” (lower right shows flow vectors against a dark background for increased visibility)

Horn-Schunck method

The method of Horn and Schunck was developed in 1981 [Horn81]. This technique was one of the first to make use of the brightness constancy assumption and to derive the basic brightness constancy equations. The solution of these equations devised by Horn and Schunck was by hypothesizing a smoothness constraint on the velocities v_x and v_y . This constraint was derived by minimizing the regularized Laplacian of the optical flow velocity components:

$$\frac{\partial}{\partial x} \frac{\partial v_x}{\partial x} - \frac{1}{\alpha} I_x (I_x v_x + I_y v_y + I_t) = 0$$

$$\frac{\partial}{\partial y} \frac{\partial v_y}{\partial y} - \frac{1}{\alpha} I_y (I_x v_x + I_y v_y + I_t) = 0$$

Here α is a constant weighting coefficient known as the *regularization constant*. Larger values of α lead to smoother (i.e., more locally consistent) vectors of motion flow. This is a relatively simple constraint for enforcing smoothness, and its effect is to penalize regions in which the flow is changing in magnitude. As with Lucas-Kanade, the Horn-Schunck technique relies on iterations to solve the differential equations. The function that computes this is:

```
void cvCalcOpticalFlowHS(
    const CvArr*    imgA,
    const CvArr*    imgB,
    int             usePrevious,
    CvArr*          velx,
```



```

        CvArr*          vely,
        double          lambda,
        CvTermCriteria  criteria
    );

```

Here `imgA` and `imgB` must be 8-bit, single-channel images. The x and y velocity results will be stored in `velx` and `vely`, which must be 32-bit, floating-point, single-channel images. The `usePrevious` parameter tells the algorithm to use the `velx` and `vely` velocities computed from a previous frame as the initial starting point for computing the new velocities. The parameter `lambda` is a weight related to the *Lagrange multiplier*. You are probably asking yourself: “What Lagrange multiplier?”* The Lagrange multiplier arises when we attempt to minimize (simultaneously) both the motion-brightness equation and the smoothness equations; it represents the relative weight given to the errors in each as we minimize.

Block matching method

You might be thinking: “What’s the big deal with optical flow? Just match where pixels in one frame went to in the next frame.” This is exactly what others have done. The term “block matching” is a catchall for a whole class of similar algorithms in which the image is divided into small regions called *blocks* [Huang95; Beauchemin95]. Blocks are typically square and contain some number of pixels. These blocks may overlap and, in practice, often do. Block-matching algorithms attempt to divide both the previous and current images into such blocks and then compute the motion of these blocks. Algorithms of this kind play an important role in many video compression algorithms as well as in optical flow for computer vision.

Because block-matching algorithms operate on aggregates of pixels, not on individual pixels, the returned “velocity images” are typically of lower resolution than the input images. This is not always the case; it depends on the severity of the overlap between the blocks. The size of the result images is given by the following formula:

$$W_{\text{result}} = \left\lfloor \frac{W_{\text{prev}} - W_{\text{block}} + W_{\text{shiftsize}}}{W_{\text{shiftsize}}} \right\rfloor_{\text{floor}}$$

$$H_{\text{result}} = \left\lfloor \frac{H_{\text{prev}} - H_{\text{block}} + H_{\text{shiftsize}}}{H_{\text{shiftsize}}} \right\rfloor_{\text{floor}}$$

The implementation in OpenCV uses a spiral search that works out from the location of the original block (in the previous frame) and compares the candidate new blocks with the original. This comparison is a sum of absolute differences of the pixels (i.e., an L1 distance). If a good enough match is found, the search is terminated. Here’s the function prototype:

* You might even be asking yourself: “What is a Lagrange multiplier?”. In that case, it may be best to ignore this part of the paragraph and just set `lambda` equal to 1.

```

void cvCalcOpticalFlowBM(
    const CvArr* prev,
    const CvArr* curr,
    CvSize      block_size,
    CvSize      shift_size,
    CvSize      max_range,
    int         use_previous,
    CvArr*      velx,
    CvArr*      vely
);

```

The arguments are straightforward. The `prev` and `curr` parameters are the previous and current images; both should be 8-bit, single-channel images. The `block_size` is the size of the block to be used, and `shift_size` is the step size between blocks (this parameter controls whether—and, if so, by how much—the blocks will overlap). The `max_range` parameter is the size of the region around a given block that will be searched for a corresponding block in the subsequent frame. If set, `use_previous` indicates that the values in `velx` and `vely` should be taken as starting points for the block searches.* Finally, `velx` and `vely` are themselves 32-bit single-channel images that will store the computed motions of the blocks. As mentioned previously, motion is computed at a block-by-block level and so the coordinates of the result images are for the blocks (i.e., aggregates of pixels), not for the individual pixels of the original image.

Mean-Shift and Camshift Tracking

In this section we will look at two techniques, *mean-shift* and *camshift* (where “cam-shift” stands for “continuously adaptive mean-shift”). The former is a general technique for data analysis (discussed in Chapter 9 in the context of segmentation) in many applications, of which computer vision is only one. After introducing the general theory of mean-shift, we’ll describe how OpenCV allows you to apply it to tracking in images. The latter technique, *camshift*, builds on mean-shift to allow for the tracking of objects whose size may change during a video sequence.

Mean-Shift

The mean-shift algorithm[†] is a robust method of finding local extrema in the density distribution of a data set. This is an easy process for continuous distributions; in that context, it is essentially just *hill climbing* applied to a density histogram of the data.[‡] For discrete data sets, however, this is a somewhat less trivial problem.

* If `use_previous==0`, then the search for a block will be conducted over a region of `max_range` distance from the location of the original block. If `use_previous!=0`, then the center of that search is first displaced by $\Delta x = \text{vel}_x(x, y)$ and $\Delta y = \text{vel}_y(x, y)$.

† Because mean-shift is a fairly deep topic, our discussion here is aimed mainly at developing intuition for the user. For the original formal derivation, see Fukunaga [Fukunaga90] and Comaniciu and Meer [Comaniciu99].

‡ The word “essentially” is used because there is also a scale-dependent aspect of mean-shift. To be exact: mean-shift is equivalent in a continuous distribution to first convolving with the mean-shift kernel and then applying a hill-climbing algorithm.

The descriptor “robust” is used here in its formal statistical sense; that is, mean-shift ignores outliers in the data. This means that it ignores data points that are far away from peaks in the data. It does so by processing only those points within a local window of the data and then moving that window.

The mean-shift algorithm runs as follows.

1. Choose a search window:
 - its initial location;
 - its type (uniform, polynomial, exponential, or Gaussian);
 - its shape (symmetric or skewed, possibly rotated, rounded or rectangular);
 - its size (extent at which it rolls off or is cut off).
2. Compute the window’s (possibly weighted) center of mass.
3. Center the window at the center of mass.
4. Return to step 2 until the window stops moving (it always will).*

To give a little more formal sense of what the mean-shift algorithm is: it is related to the discipline of *kernel density estimation*, where by “kernel” we refer to a function that has mostly local focus (e.g., a Gaussian distribution). With enough appropriately weighted and sized kernels located at enough points, one can express a distribution of data entirely in terms of those kernels. Mean-shift diverges from kernel density estimation in that it seeks only to estimate the gradient (direction of change) of the data distribution. When this change is 0, we are at a stable (though perhaps local) peak of the distribution. There might be other peaks nearby or at other scales.

Figure 10-11 shows the equations involved in the mean-shift algorithm. These equations can be simplified by considering a *rectangular kernel*,[†] which reduces the mean-shift vector equation to calculating the center of mass of the image pixel distribution:

$$x_c = \frac{M_{10}}{M_{00}}, \quad y_c = \frac{M_{01}}{M_{00}}$$

Here the zeroth moment is calculated as:

$$M_{00} = \sum_x \sum_y I(x, y)$$

and the first moments are:

* Iterations are typically restricted to some maximum number or to some epsilon change in center shift between iterations; however, they are guaranteed to converge eventually.

† A *rectangular kernel* is a kernel with no falloff with distance from the center, until a single sharp transition to zero value. This is in contrast to the exponential falloff of a Gaussian kernel and the falloff with the square of distance from the center in the commonly used Epanechnikov kernel.

$$M_{10} = \sum_x \sum_y x I(x, y) \quad \text{and} \quad M_{01} = \sum_x \sum_y y I(x, y)$$

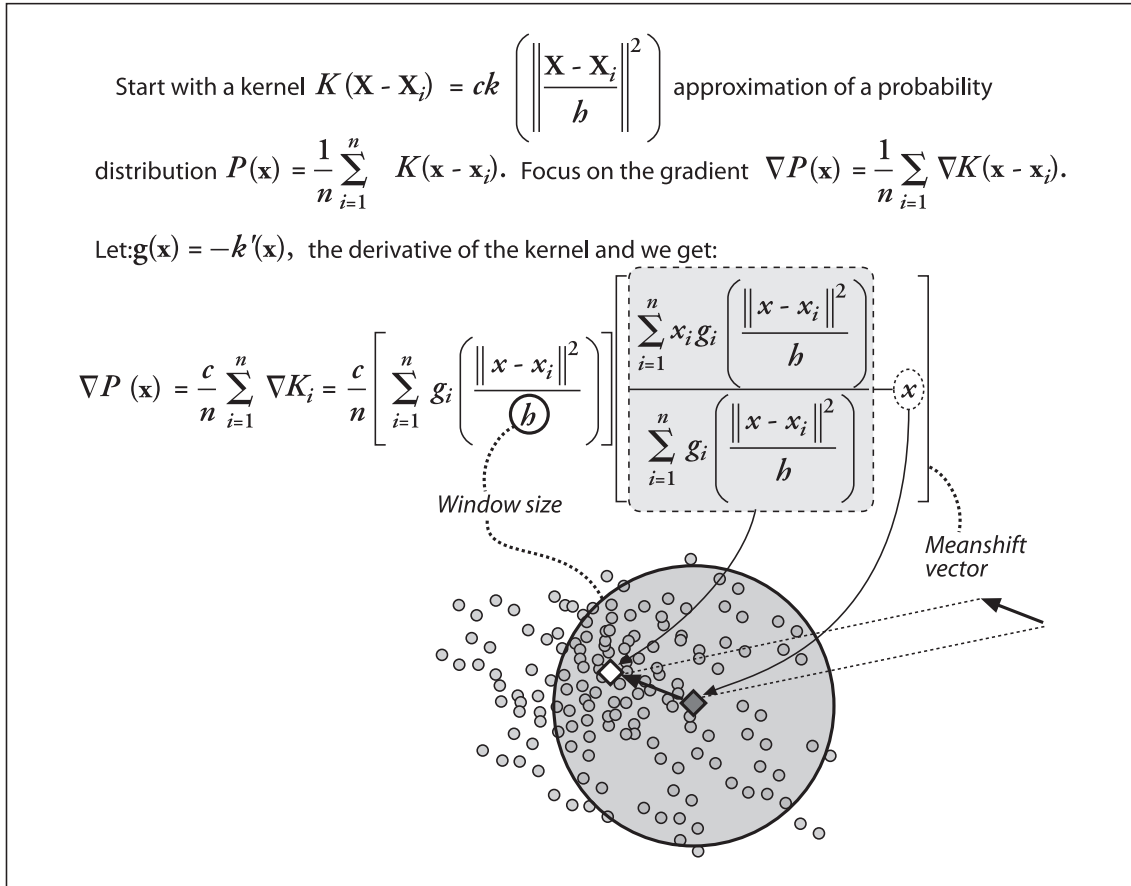


Figure 10-11. Mean-shift equations and their meaning

The mean-shift vector in this case tells us to recenter the mean-shift window over the calculated center of mass within that window. This movement will, of course, change what is “under” the window and so we iterate this recentering process. Such recentering will always converge to a mean-shift vector of 0 (i.e., where no more centering movement is possible). The location of convergence is at a local maximum (peak) of the distribution under the window. Different window sizes will find different peaks because “peak” is fundamentally a scale-sensitive construct.

In Figure 10-12 we see an example of a two-dimensional distribution of data and an initial (in this case, rectangular) window. The arrows indicate the process of convergence on a local mode (peak) in the distribution. Observe that, as promised, this peak finder is statistically robust in the sense that points outside the mean-shift window do not affect convergence—the algorithm is not “distracted” by far-away points.

In 1998, it was realized that this mode-finding algorithm could be used to track moving objects in video [Bradski98a; Bradski98b], and the algorithm has since been greatly extended [Comaniciu03]. The OpenCV function that performs mean-shift is implemented in the context of image analysis. This means in particular that, rather than taking some

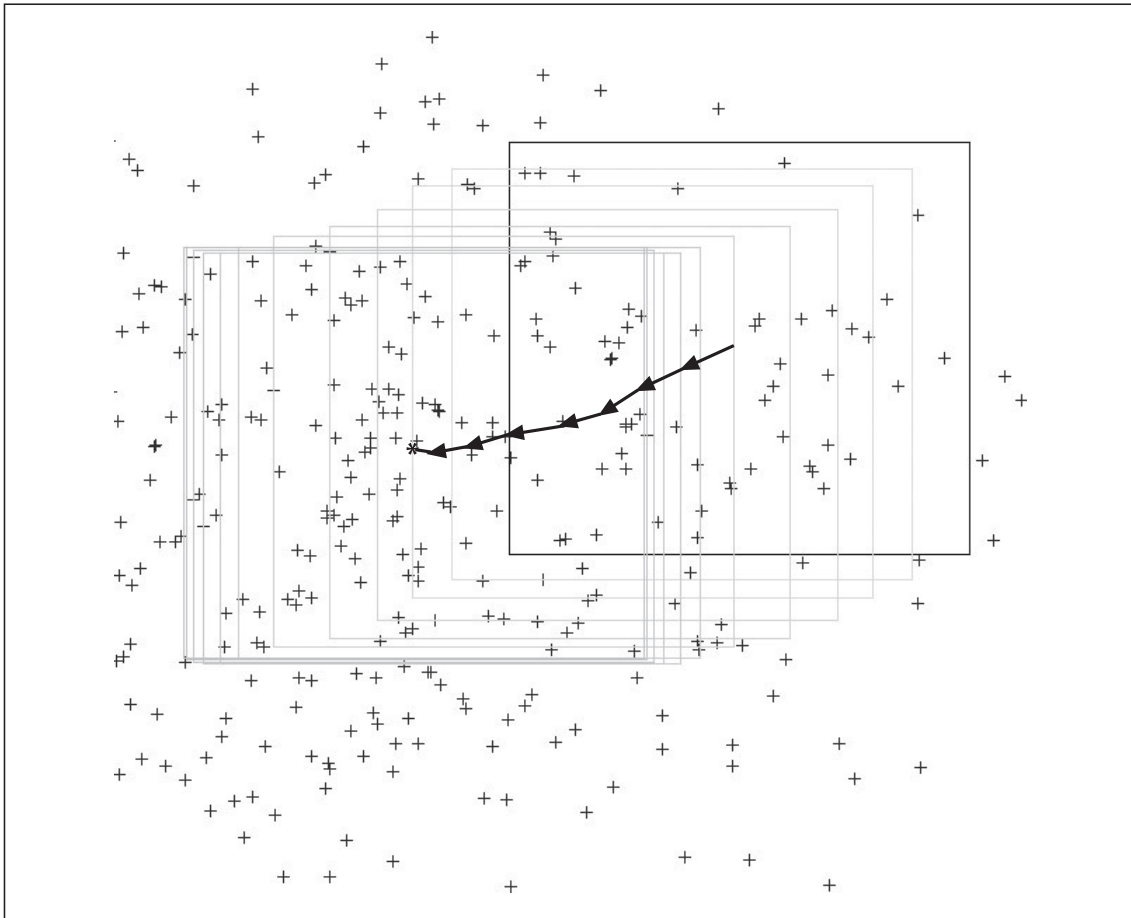


Figure 10-12. Mean-shift algorithm in action: an initial window is placed over a two-dimensional array of data points and is successively recentered over the mode (or local peak) of its data distribution until convergence

arbitrary set of data points (possibly in some arbitrary number of dimensions), the OpenCV implementation of mean-shift expects as input an image representing the density distribution being analyzed. You could think of this image as a two-dimensional histogram measuring the density of points in some two-dimensional space. It turns out that, for vision, this is precisely what you want to do most of the time: it's how you can track the motion of a cluster of interesting features.

```
int cvMeanShift(
    const CvArr*    prob_image,
    CvRect          window,
    CvTermCriteria  criteria,
    CvConnectedComp* comp
);
```

In `cvMeanShift()`, the `prob_image`, which represents the density of probable locations, may be only one channel but of either type (byte or float). The window is set at the initial desired location and size of the kernel window. The termination criteria has been described elsewhere and consists mainly of a maximum limit on number of mean-shift movement iterations and a minimal movement for which we consider the window

locations to have converged.* The connected component `comp` contains the converged search window location in `comp->rect`, and the sum of all pixels under the window is kept in the `comp->area` field.

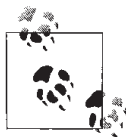
The function `cvMeanShift()` is one expression of the mean-shift algorithm for rectangular windows, but it may also be used for tracking. In this case, you first choose the feature distribution to represent an object (e.g., color + texture), then start the mean-shift window over the feature distribution generated by the object, and finally compute the chosen feature distribution over the next video frame. Starting from the current window location, the mean-shift algorithm will find the new peak or mode of the feature distribution, which (presumably) is centered over the object that produced the color and texture in the first place. In this way, the mean-shift window tracks the movement of the object frame by frame.

Camshift

A related algorithm is the Camshift tracker. It differs from the meanshift in that the search window adjusts itself in size. If you have well-segmented distributions (say face features that stay compact), then this algorithm will automatically adjust itself for the size of face as the person moves closer to and further from the camera. The form of the Camshift algorithm is:

```
int cvCamShift(  
    const CvArr*      prob_image,  
    CvRect            window,  
    CvTermCriteria     criteria,  
    CvConnectedComp*  comp,  
    CvBox2D*           box          = NULL  
);
```

The first four parameters are the same as for the `cvMeanShift()` algorithm. The `box` parameter, if present, will contain the newly resized box, which also includes the orientation of the object as computed via second-order moments. For tracking applications, we would use the resulting resized box found on the previous frame as the window in the next frame.



Many people think of mean-shift and camshift as tracking using color features, but this is not entirely correct. Both of these algorithms track the distribution of any kind of feature that is expressed in the `prob_image`; hence they make for very lightweight, robust, and efficient trackers.

Motion Templates

Motion templates were invented in the MIT Media Lab by Bobick and Davis [Bobick96; Davis97] and were further developed jointly with one of the authors [Davis99; Bradski00]. This more recent work forms the basis for the implementation in OpenCV.

* Again, mean-shift will always converge, but convergence may be very slow near the local peak of a distribution if that distribution is fairly “flat” there.

Motion templates are an effective way to track general movement and are especially applicable to gesture recognition. Using motion templates requires a silhouette (or part of a silhouette) of an object. Object silhouettes can be obtained in a number of ways.

1. The simplest method of obtaining object silhouettes is to use a reasonably stationary camera and then employ frame-to-frame differencing (as discussed in Chapter 9). This will give you the moving edges of objects, which is enough to make motion templates work.
2. You can use chroma keying. For example, if you have a known background color such as bright green, you can simply take as foreground anything that is not bright green.
3. Another way (also discussed in Chapter 9) is to learn a background model from which you can isolate new foreground objects/people as silhouettes.
4. You can use active silhouetting techniques—for example, creating a wall of near-infrared light and having a near-infrared-sensitive camera look at the wall. Any intervening object will show up as a silhouette.
5. You can use thermal imagers; then any hot object (such as a face) can be taken as foreground.
6. Finally, you can generate silhouettes by using the segmentation techniques (e.g., pyramid segmentation or mean-shift segmentation) described in Chapter 9.

For now, assume that we have a good, segmented object silhouette as represented by the white rectangle of Figure 10-13(A). Here we use white to indicate that all the pixels are set to the floating-point value of the most recent system time stamp. As the rectangle moves, new silhouettes are captured and overlaid with the (new) current time stamp; the new silhouette is the white rectangle of Figure 10-13(B) and Figure 10-13(C). Older motions are shown in Figure 10-13 as successively darker rectangles. These sequentially fading silhouettes record the history of previous movement and thus are referred to as the “motion history image”.

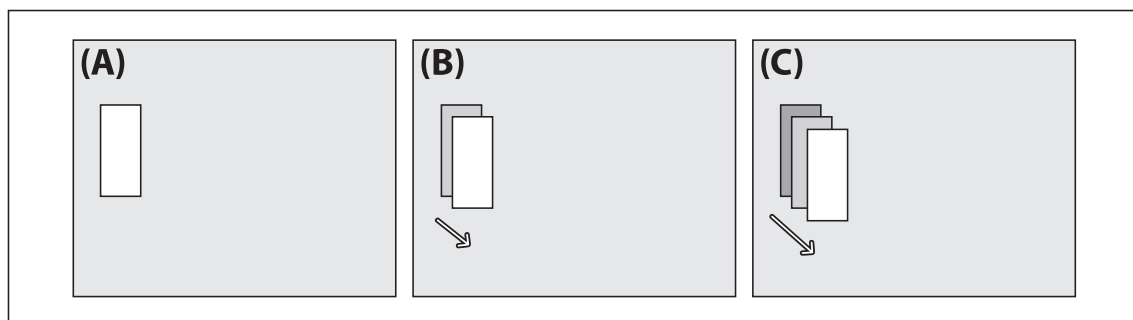


Figure 10-13. Motion template diagram: (A) a segmented object at the current time stamp (white); (B) at the next time step, the object moves and is marked with the (new) current time stamp, leaving the older segmentation boundary behind; (C) at the next time step, the object moves further, leaving older segmentations as successively darker rectangles whose sequence of encoded motion yields the motion history image

Silhouettes whose time stamp is more than a specified duration older than the current system time stamp are set to 0, as shown in Figure 10-14. The OpenCV function that accomplishes this motion template construction is `cvUpdateMotionHistory()`:

```
void cvUpdateMotionHistory(
    const CvArr* silhouette,
    CvArr*      mhi,
    double      timestamp,
    double      duration
);
```

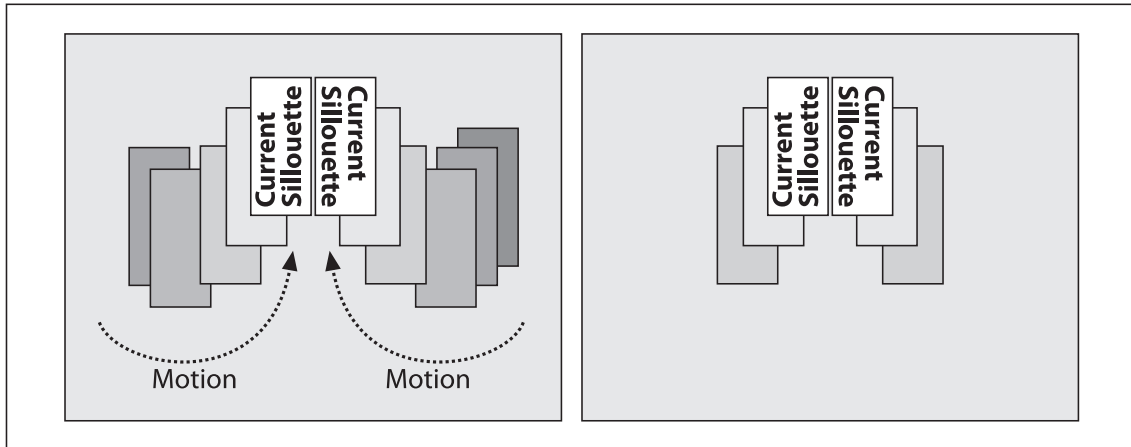


Figure 10-14. Motion template silhouettes for two moving objects (left); silhouettes older than a specified duration are set to 0 (right)

In `cvUpdateMotionHistory()`, all image arrays consist of single-channel images. The silhouette image is a byte image in which nonzero pixels represent the most recent segmentation silhouette of the foreground object. The `mhi` image is a floating-point image that represents the motion template (aka motion history image). Here `timestamp` is the current system time (typically a millisecond count) and `duration`, as just described, sets how long motion history pixels are allowed to remain in the `mhi`. In other words, any `mhi` pixels that are older (less) than `timestamp` minus `duration` are set to 0.

Once the motion template has a collection of object silhouettes overlaid in time, we can derive an indication of overall motion by taking the gradient of the `mhi` image. When we take these gradients (e.g., by using the Scharr or Sobel gradient functions discussed in Chapter 6), some gradients will be large and invalid. Gradients are invalid when older or inactive parts of the `mhi` image are set to 0, which produces artificially large gradients around the outer edges of the silhouettes; see Figure 10-15(A). Because we know the time-step duration with which we've been introducing new silhouettes into the `mhi` via `cvUpdateMotionHistory()`, we know how large our gradients (which are just dx and dy step derivatives) should be. We can therefore use the gradient magnitude to eliminate gradients that are too large, as in Figure 10-15(B). Finally, we can collect a measure of global motion; see Figure 10-15(C). The function that effects parts (A) and (B) of the figure is `cvCalcMotionGradient()`:

```

void cvCalcMotionGradient(
    const CvArr* mhi,
    CvArr* mask,
    CvArr* orientation,
    double delta1,
    double delta2,
    int aperture_size=3
);

```

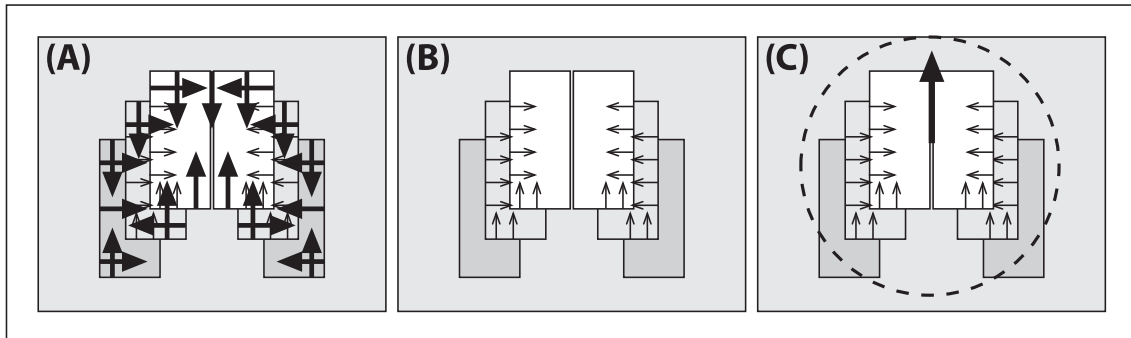


Figure 10-15. Motion gradients of the mhi image: (A) gradient magnitudes and directions; (B) large gradients are eliminated; (C) overall direction of motion is found

In `cvCalcMotionGradient()`, all image arrays are single-channel. The function input `mhi` is a floating-point motion history image, and the input variables `delta1` and `delta2` are (respectively) the minimal and maximal gradient magnitudes allowed. Here, the expected gradient magnitude will be just the average number of time-stamp ticks between each silhouette in successive calls to `cvUpdateMotionHistory()`; setting `delta1` halfway below and `delta2` halfway above this average value should work well. The variable `aperture_size` sets the size in width and height of the gradient operator. These values can be set to -1 (the 3-by-3 `CV_SCHARR` gradient filter), 3 (the default 3-by-3 Sobel filter), 5 (for the 5-by-5 Sobel filter), or 7 (for the 7-by-7 filter). The function outputs are `mask`, a single-channel 8-bit image in which nonzero entries indicate where valid gradients were found, and `orientation`, a floating-point image that gives the gradient direction's angle at each point.

The function `cvCalcGlobalOrientation()` finds the overall direction of motion as the vector sum of the valid gradient directions.

```

double cvCalcGlobalOrientation(
    const CvArr* orientation,
    const CvArr* mask,
    const CvArr* mhi,
    double      timestamp,
    double      duration
);

```

When using `cvCalcGlobalOrientation()`, we pass in the `orientation` and `mask` image computed in `cvCalcMotionGradient()` along with the `timestamp`, `duration`, and resulting `mhi` from `cvUpdateMotionHistory()`; what's returned is the vector-sum global orientation,

as in Figure 10-15(C). The timestamp together with duration tells the routine how much motion to consider from the mhi and motion orientation images. One could compute the global motion from the center of mass of each of the mhi silhouettes, but summing up the precomputed motion vectors is much faster.

We can also isolate regions of the motion template mhi image and determine the local motion within that region, as shown in Figure 10-16. In the figure, the mhi image is scanned for current silhouette regions. When a region marked with the most current time stamp is found, the region's perimeter is searched for sufficiently recent motion (recent silhouettes) just outside its perimeter. When such motion is found, a downward-stepping flood fill is performed to isolate the local region of motion that "spilled off" the current location of the object of interest. Once found, we can calculate local motion gradient direction in the spill-off region, then remove that region, and repeat the process until all regions are found (as diagrammed in Figure 10-16).

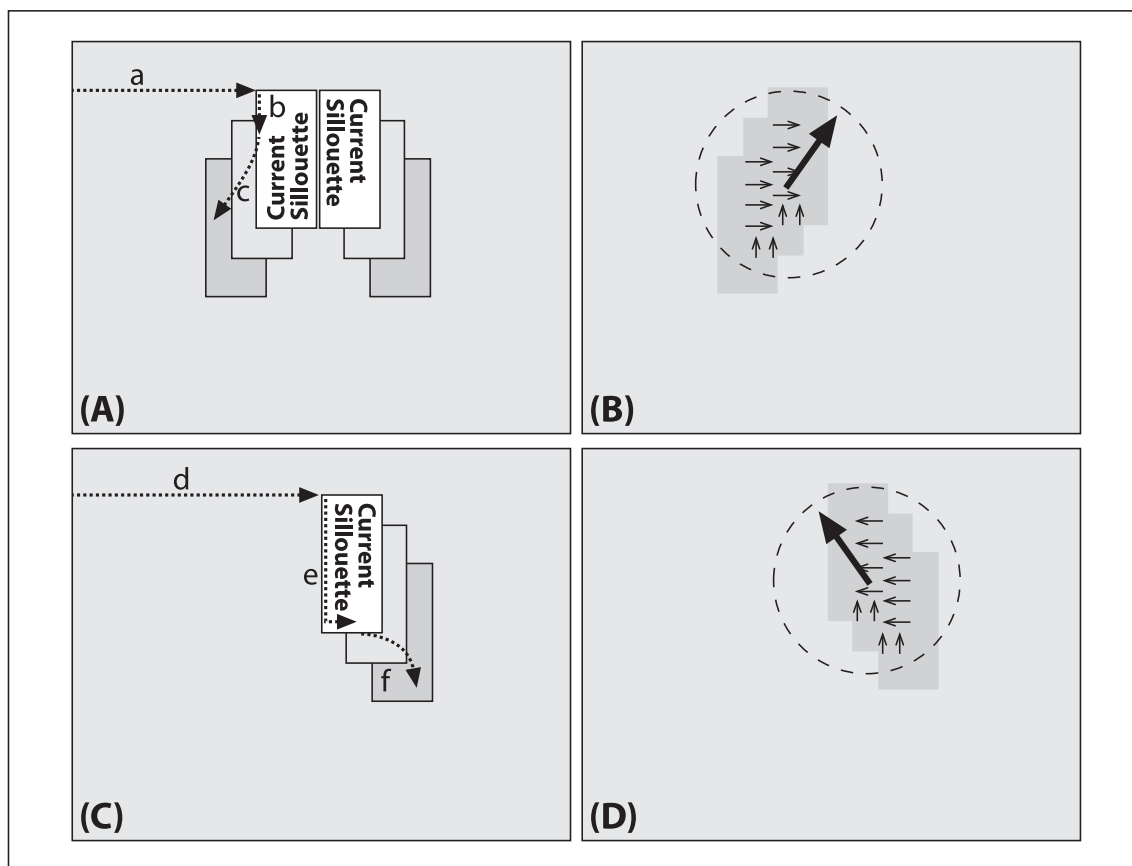


Figure 10-16. Segmenting local regions of motion in the mhi image: (A) scan the mhi image for current silhouettes (a) and, when found, go around the perimeter looking for other recent silhouettes (b); when a recent silhouette is found, perform downward-stepping flood fills (c) to isolate local motion; (B) use the gradients found within the isolated local motion region to compute local motion; (C) remove the previously found region and search for the next current silhouette region (d), scan along it (e), and perform downward-stepping flood fill on it (f); (D) compute motion within the newly isolated region and continue the process (A)-(C) until no current silhouette remains

The function that isolates and computes local motion is `cvSegmentMotion()`:

```
CvSeq* cvSegmentMotion(  
    const CvArr*  mhi,  
    CvArr*        seg_mask,  
    CvMemStorage* storage,  
    double        timestamp,  
    double        seg_thresh  
);
```

In `cvSegmentMotion()`, the `mhi` is the single-channel floating-point input. We also pass in `storage`, a `CvMemoryStorage` structure allocated via `cvCreateMemStorage()`. Another input is `timestamp`, the value of the most current silhouettes in the `mhi` from which you want to segment local motions. Finally, you must pass in `seg_thresh`, which is the maximum downward step (from current time to previous motion) that you'll accept as attached motion. This parameter is provided because there might be overlapping silhouettes from recent and much older motion that you don't want to connect together.

It's generally best to set `seg_thresh` to something like 1.5 times the average difference in silhouette time stamps. This function returns a `CvSeq` of `CvConnectedComp` structures, one for each separate motion found, which delineates the local motion regions; it also returns `seg_mask`, a single-channel, floating-point image in which each region of isolated motion is marked a distinct nonzero number (a zero pixel in `seg_mask` indicates no motion). To compute these local motions one at a time we call `cvCalcGlobalOrientation()`, using the appropriate mask region selected from the appropriate `CvConnectedComp` or from a particular value in the `seg_mask`; for example,

```
cvCmpS(  
    seg_mask,  
    // [value_wanted_in_seg_mask],  
    // [your_destination_mask],  
    CV_CMP_EQ  
)
```

Given the discussion so far, you should now be able to understand the *motempl.c* example that ships with OpenCV in the `.../opencv/samples/c/` directory. We will now extract and explain some key points from the `update_mhi()` function in *motempl.c*. The `update_mhi()` function extracts templates by thresholding frame differences and then passing the resulting silhouette to `cvUpdateMotionHistory()`:

```
...  
cvAbsDiff( buf[idx1], buf[idx2], silh );  
cvThreshold( silh, silh, diff_threshold, 1, CV_THRESH_BINARY );  
cvUpdateMotionHistory( silh, mhi, timestamp, MHI_DURATION );  
...
```

The gradients of the resulting `mhi` image are then taken, and a mask of valid gradients is produced using `cvCalcMotionGradient()`. Then `CvMemStorage` is allocated (or, if it already exists, it is cleared), and the resulting local motions are segmented into `CvConnectedComp` structures in the `CvSeq` containing structure `seq`:

```
...  
cvCalcMotionGradient(  
    ...  
);
```

```

    mhi,
    mask,
    orient,
    MAX_TIME_DELTA,
    MIN_TIME_DELTA,
    3
);

if( !storage )
    storage = cvCreateMemStorage(0);
else
    cvClearMemStorage(storage);

seq = cvSegmentMotion(
    mhi,
    segmask,
    storage,
    timestamp,
    MAX_TIME_DELTA
);

```

A “for” loop then iterates through the `seq->total` `CvConnectedComp` structures extracting bounding rectangles for each motion. The iteration starts at -1, which has been designated as a special case for finding the global motion of the whole image. For the local motion segments, small segmentation areas are first rejected and then the orientation is calculated using `cvCalcGlobalOrientation()`. Instead of using exact masks, this routine restricts motion calculations to regions of interest (ROIs) that bound the local motions; it then calculates where valid motion within the local ROIs was actually found. Any such motion area that is too small is rejected. Finally, the routine draws the motion. Examples of the output for a person flapping their arms is shown in Figure 10-17, where the output is drawn above the raw image for four sequential frames going across in two rows. (For the full code, see `.../opencv/samples/c/motempl.c`.) In the same sequence, “Y” postures were recognized by the shape descriptors (Hu moments) discussed in Chapter 8, although the shape recognition is not included in the *samples* code.

```

for( i = -1; i < seq->total; i++ ) {
    if( i < 0 ) { // case of the whole image
//        ...[does the whole image]...
    else { // i-th motion component
        comp_rect = ((CvConnectedComp*)cvGetSeqElem( seq, i ))->rect;
//        [reject very small components]...
    }
    ...[set component ROI regions]...
    angle = cvCalcGlobalOrientation( orient, mask, mhi,
                                    timestamp, MHI_DURATION);
    ...[find regions of valid motion]...
    ...[reset ROI regions]...
    ...[skip small valid motion regions]...
    ...[draw the motions]...
}

```




Figure 10-17. Results of motion template routine: going across and top to bottom, a person moving and the resulting global motions indicated in large octagons and local motions indicated in small octagons; also, the “Y” pose can be recognized via shape descriptors (Hu moments)

Estimators

Suppose we are tracking a person who is walking across the view of a video camera. At each frame we make a determination of the location of this person. This could be done any number of ways, as we have seen, but in each case we find ourselves with an estimate of the position of the person at each frame. This estimation is not likely to be