24

Non-Rigid Image Matching

The correlation-based registration methods described in Chapter 23 are *rigid* in the sense that they provide for *translation* as the only form of geometric transformation and positioning is limited to whole pixel units. In this chapter we look at methods that are capable of registering a reference image under (almost) arbitrary geometric transformations, such as changes in rotation, scale, and affine distortion, and also to *sub-pixel* accuracy.

At the core of this chapter is a detailed description of the classic Lucas-Kanade algorithm [154] and its efficient implementation. Unlike the methods presented earlier, the algorithms described here typically do not perform a global search over the entire image to find the best match, but start from an initial estimate of the geometric transformation to home in on the optimum position and distortion in an iterative fashion. This is not difficult, for example, in tracking applications, where the approximate location of a particular image patch can be predicted from the observed motion in previous frames. Of course, the global matching methods described in Chapter 23 can be used to find a coarse starting solution.

24.1 The Lucas-Kanade Technique

The basic idea of the Lucas-Kanade technique is best illustrated in the 1D case (see Fig. 24.1(a)).

24.1.1 Registration in 1D

Given two 1D, real-valued functions $f(x)$, $g(x)$, the registration problem is to find the disparity t in the (horizontal) x-direction under the assumption that g is a shifted version of f, that is,

$$g(x) = f(x - t). \tag{24.1}$$

If the function f is linear in a (sufficiently large) neighborhood of some point x with slope $f'(x)$, then

© Springer-Verlag London 2016
W. Burger, M.J. Burge, *Digital Image Processing*, Texts in Computer Science,
DOI 10.1007/978-1-4471-6684-9_24

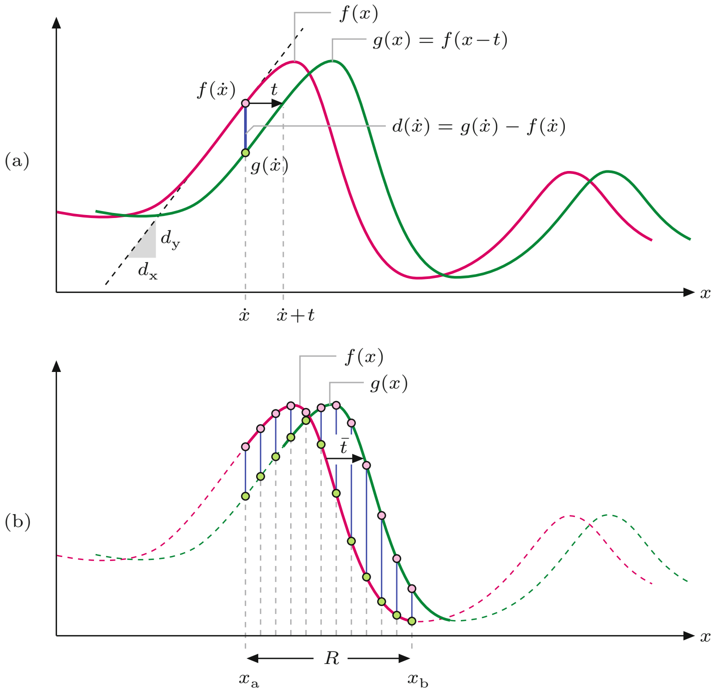

Fig. 24.1
Registering two 1D functions (figure adapted from [154]). The 1D function $g(x)$ is assumed to be a shifted version of $f(x)$. In (a), f is approximately linear at position $\dot{x}$, with slope $f'(\dot{x}) = d_y/d_x$. Under this condition, the horizontal displacement t can be estimated from the difference of the local function values $f(\dot{x})$ and $g(\dot{x})$ as $t \approx (f(\dot{x}) - g(\dot{x}))/f'(\dot{x})$. In (b), the overall displacement $\bar{t}$ is calculated by averaging the individual displacement estimates from multiple samples in the region $R = [x_a, x_b]$.

$$f(x-t) \approx f(x) - t \cdot f'(x) \tag{24.2}$$

and therefore

$$g(x) \approx f(x) - t \cdot f'(x). \tag{24.3}$$

Thus, given the function values $f(x)$, $g(x)$ and the first derivative $f'(x)$ at some point x, the displacement t can be estimated (from Eqn. (24.2)) as

$$t \approx \frac{f(x) - g(x)}{f'(x)}. \tag{24.4}$$

Note that this can be viewed as a first-order Taylor expansion[1] of the function f. Obviously, the estimate of the shift t in Eqn. (24.4) depends only on a single pair of function samples at position x and fails at points where f is either not linear or flat, that is, where the first derivative f' vanishes. To obtain a more robust displacement estimate it appears natural to extend the calculation over a range R of sample values, thereby aligning a complete section of the two functions f and g (see Fig. 24.1(b)). This problem can be formulated as finding the displacement t that minimizes the L_2 distance between the two functions f and g over a range R, that is, finding t such that

$$\mathcal{E}(t) = \sum_{x \in R} [f(x-t) - g(x)]^2 = \sum_{x \in R} [f(x) - t \cdot f'(x) - g(x)]^2 \tag{24.5}$$

[1] See also Sec. C.3.2 in the Appendix.

is a minimum. This can be accomplished by calculating the first derivative of the aforementioned expression (with respect to t) and setting it equal to zero, which gives

$$\frac{\partial \mathcal{E}}{\partial t} = 2 \cdot \sum_{x \in R} f'(x) \cdot \big[f(x) - f'(x) \cdot t - g(x) \big] = 0 \,. \tag{24.6}$$

By solving this equation the optimal shift is found as

$$t_{\text{opt}} = \Big[\sum_{x \in R} [f'(x)]^2 \Big]^{-1} \cdot \sum_{x \in R} f'(x) \cdot [f(x) - g(x)] \,. \tag{24.7}$$

Note that this local estimation works even if the function f is flat at some positions in R, unless $f'(x)$ is zero everywhere R. However, since the estimate is based only on linear (i.e., first-order) prediction, the estimate is generally not accurate. For this purpose, the following iterative optimization scheme is proposed in [154], which is really the basis of the Lucas-Kanade algorithm. With $t^{(0)} = t_{\text{start}}$ as the initial estimate of the displacement (which may be zero), t is successively updated as

$$t^{(k)} = t^{(k-1)} + \Big[\sum_{x \in R} [f'(x)]^2 \Big]^{-1} \cdot \sum_{x \in R} f'(x) \cdot [f(x) - g(x)] \,, \tag{24.8}$$

for $k = 1, 2, \ldots$, until either $t^{(k)}$ converges or a maximum number of steps is reached.

24.1.2 Extension to Multi-Dimensional Functions

As shown in [154], the formulation given in Sec. 24.1.1 can be easily generalized to align multi-dimensional, scalar-valued functions, including 2D images. In general, the involved functions $F(\boldsymbol{x})$ and $G(\boldsymbol{x})$ are now defined over $\mathbb{R}^m$, and thus all coordinates $\boldsymbol{x} = (x_1, \ldots, x_m)$ and spatial shifts $\boldsymbol{t} = (t_1, \ldots, t_m)$ are m-dimensional column vectors. The task is, analogous to Eqn. (24.5), to find the vector $\boldsymbol{t}$ that minimizes the error quantity

$$\mathcal{E}(\boldsymbol{t}) = \sum_{\boldsymbol{x} \in R} \big[F(\boldsymbol{x} - \boldsymbol{t}) - G(\boldsymbol{x}) \big]^2, \tag{24.9}$$

where R denotes an m-dimensional region. The linear approximation in Eqn. (24.2) becomes

$$F(\boldsymbol{x} - \boldsymbol{t}) \approx F(\boldsymbol{x}) - \nabla_F(\boldsymbol{x}) \cdot \boldsymbol{t}, \tag{24.10}$$

where the row vector $\nabla_F(\boldsymbol{x}) = \big(\frac{\partial F}{\partial x_1}(\boldsymbol{x}), \ldots, \frac{\partial F}{\partial x_m}(\boldsymbol{x}) \big)$ is the m-dimensional *gradient* of the function F, evaluated at position $\boldsymbol{x}$. Minimizing $\mathcal{E}(\boldsymbol{t})$ over $\boldsymbol{t}$ is again accomplished by solving $\frac{\partial \mathcal{E}}{\partial \boldsymbol{t}} = 0$, that is (analogous to Eqn. (24.6)),

$$2 \cdot \sum_{\boldsymbol{x} \in R} \nabla_F(\boldsymbol{x}) \cdot \big[F(\boldsymbol{x}) - \nabla_F(\boldsymbol{x}) \cdot \boldsymbol{t} - G(\boldsymbol{x}) \big] = 0 \,. \tag{24.11}$$

The solution to Eqn. (24.11) is

$$\boldsymbol{t}_{\mathrm{opt}} = \Big[\sum_{\boldsymbol{x}\in R} \nabla_F^{\mathsf{T}}(\boldsymbol{x}) \cdot \nabla_F(\boldsymbol{x})\Big]^{-1} \cdot \Big[\sum_{\boldsymbol{x}\in R} \nabla_F^{\mathsf{T}}(\boldsymbol{x}) \cdot \big[F(\boldsymbol{x}) - G(\boldsymbol{x})\big]\Big] \qquad (24.12)$$

$$= \mathbf{H}_F^{-1} \cdot \Big[\sum_{\boldsymbol{x}\in R} \nabla_F^{\mathsf{T}}(\boldsymbol{x}) \cdot [F(\boldsymbol{x}) - G(\boldsymbol{x})]\Big], \qquad (24.13)$$

where $\mathbf{H}_F$ is an estimate of the $m \times m$ Hessian matrix[2] for the function F over the region R. Note the similarity of Eqn. (24.13) to the 1D version in Eqn. (24.7).

24.2 The Lucas-Kanade Algorithm

Based on the ideas outlined in Sec. 24.1, the Lucas-Kanade algorithm [154] is not only capable of registering 2D images by finding the optimal translation, but works for a range of geometric transformations $T_{\boldsymbol{p}}$ that can be parameterized by a n-dimensional vector $\boldsymbol{p}$. Among others, this includes affine and projective transformations (see Ch. 21) as the most important cases.

The same mathematical notation is used as in Chapter 23, that is, I denotes the *search image* and R is the (typically smaller) *reference image*. The placement and possible distortion of the matching image patch is described by a *geometric transformation* $T_{\boldsymbol{p}}$ (cf. Ch. 21), where $\boldsymbol{p}$ denotes a vector of transformation parameters. The goal of the Lucas-Kanade registration algorithm is to minimize the expression

$$\mathcal{E}(\boldsymbol{p}) = \sum_{\boldsymbol{x}\in R} \big[I(T_{\boldsymbol{p}}(\boldsymbol{x})) - R(\boldsymbol{x})\big]^2 \qquad (24.14)$$

with respect to the geometric transformation parameters $\boldsymbol{p}$, where I is the (search) image, R is the reference image (template), and $T_{\boldsymbol{p}}(\boldsymbol{x})$ is a geometric transformation or warp function with parameters $\boldsymbol{p}$. For example, simple 2D translation is described by the transformation

$$T_{\boldsymbol{p}}(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{p} = \begin{pmatrix} x + t_{\mathrm{x}} \\ y + t_{\mathrm{y}} \end{pmatrix}, \qquad (24.15)$$

where $\boldsymbol{x} = (x, y)^{\mathsf{T}}$ and $\boldsymbol{p} = (t_{\mathrm{x}}, t_{\mathrm{y}})^{\mathsf{T}}$. The task of the alignment process is to find the parameters that describe how to warp the search image I, such that the match between I and R is optimal over the support region R. Figure 24.2 illustrates the corresponding geometry.

In each iteration, the Lucas-Kanade algorithm starts with an estimate of the transformation parameters $\boldsymbol{p}$ and attempts to find the parameter increment $\boldsymbol{q}$ that locally minimizes the expression

$$\mathcal{E}(\boldsymbol{q}) = \sum_{\boldsymbol{x}\in R} \big[I(T_{\boldsymbol{p}+\boldsymbol{q}}(\boldsymbol{x})) - R(\boldsymbol{x})\big]^2. \qquad (24.16)$$

After calculating the optimal parameter change $\boldsymbol{q}_{\mathrm{opt}}$, the parameter vector $\boldsymbol{p}$ is updated in the form

$$\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{q}_{\mathrm{opt}} \qquad (24.17)$$

[2] See Sec. C.2.6 in the Appendix for details.

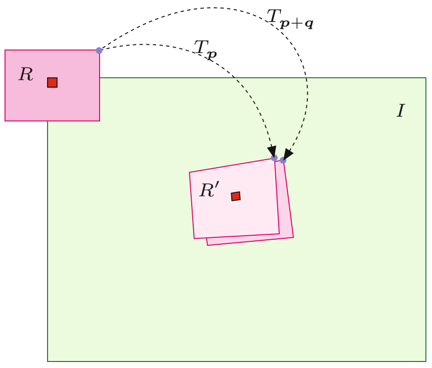

Fig. 24.2
Geometric relations in the (forward) Lucas-Kanade registration algorithm. I denotes the search image and R is the reference image. The mapping $T_{\boldsymbol{p}}$ warps the reference image R from the original position (centered at the origin) to R', with $\boldsymbol{p}$ being the initial parameter estimate. Matching is performed between the search image I and the warped reference image R'. $T_{\boldsymbol{p}+\boldsymbol{q}}$ is the improved warp; the optimal parameter change $\boldsymbol{q}$ is estimated in each iteration.

until the process converges. Typically, the update loop is terminated when the magnitude of the change vector $\boldsymbol{q}_{\text{opt}}$ drops below a predefined threshold.

The expression to be minimized in Eqn. (24.16) depends on the image content and is generally nonlinear with respect to $\boldsymbol{q}$. A locally linear approximation of this function is obtained by the first-order Taylor expansion on I, that is,[3]

$$I(T_{\boldsymbol{p}+\boldsymbol{q}}(\boldsymbol{x})) \approx I(T_{\boldsymbol{p}}(\boldsymbol{x})) + \underbrace{\underbrace{\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{x}))}_{1\times 2} \cdot \underbrace{\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x})}_{2\times n} \cdot \underbrace{\boldsymbol{q}}_{n\times 1}}_{\in \mathbb{R}}, \tag{24.18}$$

where the 2D (column) vector

$$\nabla_I(\boldsymbol{x}) = \big(I_{\mathrm{x}}(\boldsymbol{x}), I_{\mathrm{y}}(\boldsymbol{x})\big) \tag{24.19}$$

is the *gradient* of the image I at some position $\boldsymbol{x}$ and $\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x})$ denotes the *Jacobian* matrix[4] of the warp function $T_{\boldsymbol{p}}$, also evaluated at position $\boldsymbol{x}$. In general, the Jacobian of a 2D warp function

$$T_{\boldsymbol{p}}(\boldsymbol{x}) = \begin{pmatrix} T_{\mathrm{x},\boldsymbol{p}}(\boldsymbol{x}) \\ T_{\mathrm{y},\boldsymbol{p}}(\boldsymbol{x}) \end{pmatrix} \tag{24.20}$$

with n parameters $\boldsymbol{p} = (p_0, p_1, \ldots, p_{n-1})^{\mathsf{T}}$ is a $2 \times n$ matrix function

$$\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_0}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_1}(\boldsymbol{x}) & \ldots & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_{n-1}}(\boldsymbol{x}) \\ \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_0}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_1}(\boldsymbol{x}) & \ldots & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_{n-1}}(\boldsymbol{x}) \end{pmatrix}. \tag{24.21}$$

With the linear approximation in Eqn. (24.18), the original minimization problem in Eqn. (24.14) can now be written as

[3] In some of the following equations, we distinguish carefully between row and column vectors and the dimensions of vectors and matrices are explicitly displayed (in underbraces) to avoid possible confusion.

[4] The Jacobian $\mathbf{J}$ of a function f is a matrix containing the first partial derivatives of f, that is, it is a matrix of functions (see also Sec. C.2.1 in the Appendix).

$$\mathcal{E}(\boldsymbol{q}) \approx \sum_{\boldsymbol{u}\in R} \big[I(T_{\boldsymbol{p}}(\boldsymbol{u})) + \nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u})) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u}) \cdot \boldsymbol{q} - R(\boldsymbol{u})\big]^2 \tag{24.22}$$

$$= \sum_{\boldsymbol{u}\in R} \big[I(\acute{\boldsymbol{u}}) + \nabla_I(\acute{\boldsymbol{u}}) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u}) \cdot \boldsymbol{q} - R(\boldsymbol{u})\big]^2, \tag{24.23}$$

with $\acute{\boldsymbol{x}} = T_{\boldsymbol{p}}(\boldsymbol{x})$. Finding the parameters $\boldsymbol{q}$ that give the smallest difference $\mathcal{E}(\boldsymbol{q})$ is a linear least-squares minimization problem, which can be solved by taking the first partial derivative with respect to $\boldsymbol{q}$, that is,

$$\underbrace{\frac{\partial d}{\partial \boldsymbol{q}}}_{n\times 1} \approx \sum_{\boldsymbol{u}\in R} \underbrace{\big[\underbrace{\nabla_I(\acute{\boldsymbol{u}})}_{1\times 2} \cdot \underbrace{\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})}_{2\times n}\big]^{\mathsf{T}}}_{n\times 1} \cdot \underbrace{\big[I(\acute{\boldsymbol{u}}) + \underbrace{\nabla_I(\acute{\boldsymbol{u}})}_{1\times 2} \cdot \underbrace{\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})}_{2\times n} \cdot \underbrace{\boldsymbol{q}}_{n\times 1} - R(\boldsymbol{u})\big]^2}_{\in\mathbb{R}}, \tag{24.24}$$

and setting it equal to zero.[5] Solving the resulting equation for the unknown $\boldsymbol{q}$ yields the parameter change minimizing Eqn. (24.24) as

$$\boldsymbol{q}_{\text{opt}} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}}\,, \tag{24.25}$$

where $\bar{\mathbf{H}}$ is an estimate of the Hessian matrix (see Eqns. (24.29)–(24.30)),

$$\boldsymbol{\delta}_{\boldsymbol{p}} = \sum_{\boldsymbol{u}\in R} \underbrace{\big[\nabla_I(\acute{\boldsymbol{u}}) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big]^{\mathsf{T}}}_{\boldsymbol{s}(\boldsymbol{u})\,\in\,\mathbb{R}^n} \cdot \underbrace{\big[R(\boldsymbol{u}) - I(\acute{\boldsymbol{u}})\big]}_{D(\boldsymbol{u})\,\in\,\mathbb{R}} = \sum_{\boldsymbol{u}\in R} \boldsymbol{s}^{\mathsf{T}}(\boldsymbol{u}) \cdot D(\boldsymbol{u}) \tag{24.26}$$

is a n-dimensional column vector, and

$$D(\boldsymbol{u}) = R(\boldsymbol{u}) - I(\acute{\boldsymbol{u}}) \tag{24.27}$$

is the resulting (scalar-valued) error image. $\boldsymbol{s}(\boldsymbol{u}) = (s_0(\boldsymbol{u}), \ldots, s_{n-1}(\boldsymbol{u}))$ is a n-dimensional row vector, with each element corresponding to one of the parameters in $\boldsymbol{p}$. The 2D *scalar* fields formed by the individual components of the vector field $\boldsymbol{s}(\boldsymbol{u})$,

$$s_0, \ldots, s_{n-1} \colon M_R \times N_R \mapsto \mathbb{R}, \tag{24.28}$$

are called *steepest descent images* for the current transformation parameters $\boldsymbol{p}$.[6] These images are of the same size as the reference image R. Finally, the $n \times n$ matrix

$$\bar{\mathbf{H}} = \sum_{\boldsymbol{u}\in R} \underbrace{\big[\underbrace{\nabla_I(\acute{\boldsymbol{u}})}_{1\times 2} \cdot \underbrace{\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})}_{2\times n}\big]^{\mathsf{T}}}_{n\times 1} \cdot \underbrace{\big[\underbrace{\nabla_I(\acute{\boldsymbol{u}})}_{1\times 2} \cdot \underbrace{\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})}_{2\times n}\big]}_{1\times n} \tag{24.29}$$

$$= \sum_{\boldsymbol{u}\in R} \boldsymbol{s}^{\mathsf{T}}(\boldsymbol{u}) \cdot \boldsymbol{s}(\boldsymbol{u}) \;\approx\; \begin{pmatrix} \frac{\partial^2 D}{\partial p_0^2}(\boldsymbol{p}) & \cdots & \frac{\partial^2 D}{\partial p_0\, \partial p_{n-1}}(\boldsymbol{p}) \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 D}{\partial p_{n-1}\, \partial p_0}(\boldsymbol{p}) & \cdots & \frac{\partial^2 D}{\partial p_{n-1}^2}(\boldsymbol{p}) \end{pmatrix} \tag{24.30}$$

[5] Note that in Eqn. (24.24) the left factor inside the summation is a n-dimensional column vector, while the right factor is a scalar.

[6] The value $s_k(\boldsymbol{u})$ indicates the optimal change of parameter p_k for the individual pixel position $\boldsymbol{u}$ to achieve a steepest-descent optimization of Eqn. (24.23) (see [13, Sec. 4.3]).

in Eqn. (24.25) is an estimate of the Hessian matrix[7] for the given transformation parameters $\boldsymbol{p}$, calculated over all coordinates $\boldsymbol{x}$ of the reference image R (Eqn. (24.29)).

The inverse of this matrix is used to calculate the optimal parameter change $\boldsymbol{q}_{\mathrm{opt}}$ in Eqn. (24.25). A better alternative to this formulation is to solve

$$\bar{\mathbf{H}} \cdot \boldsymbol{q}_{\mathrm{opt}} = \boldsymbol{\delta}_{\boldsymbol{p}}, \tag{24.31}$$

for $\boldsymbol{q}_{\mathrm{opt}}$ as the unknown, without explicitly calculating $\mathbf{H}_{\boldsymbol{p}}^{-1}$. This is a system of linear equations in the standard form $\boldsymbol{A} \cdot \boldsymbol{x} = \boldsymbol{b}$, which is numerically more stable and efficient to solve than Eqn. (24.25).[8]

24.2.1 Summary of the Algorithm

In order not to get lost after this (quite mathematical) presentation, let us recap the key steps of the Lucas-Kanade method in a more compact form. In summary, given a search image I, a reference image R, a geometric transformation $T_{\boldsymbol{p}}$, an initial parameter estimate $\boldsymbol{p}_{\mathrm{init}}$, and the convergence limit ϵ, the Lucas-Kanade algorithm performs the following steps:

A. **Initialize:**
   1. Calculate the gradient $\nabla_I(\boldsymbol{u})$ of the search image I for all image positions $\boldsymbol{u} \in I$.
   2. Initialize the transformation parameters: $\boldsymbol{p} \leftarrow \boldsymbol{p}_{\mathrm{init}}$.

B. **Repeat:**
   3. Calculate the warped gradient image $\nabla'_I(\boldsymbol{u}) = \nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u}))$, for each position $\boldsymbol{u} \in R$ (by interpolation of ∇_I).
   4. Calculate the $(2 \times n)$ Jacobian matrix $\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u}) = \frac{\partial T_{\boldsymbol{p}}}{\partial \boldsymbol{p}}(\boldsymbol{u})$ of the warp function $T_{\boldsymbol{p}}(\boldsymbol{x})$, for each position $\boldsymbol{u} \in R$ and the current parameter vector $\boldsymbol{p}$ (see Eqn. (24.21)).
   5. Compute the n-dim. row vectors $\boldsymbol{s}_{\boldsymbol{u}} = \nabla'_I(\boldsymbol{u}) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})$, for each position $\boldsymbol{u} \in R$ (see Eqn. (24.26)).
   6. Compute the cumulative $n \times n$ Hessian matrix as $\bar{\mathbf{H}} = \sum_{\boldsymbol{u} \in R} \boldsymbol{s}_{\boldsymbol{u}}^{\mathsf{T}} \cdot \boldsymbol{s}_{\boldsymbol{u}}$ (see Eqn. (24.29)).
   7. Calculate the error image $D(\boldsymbol{x}) = R(\boldsymbol{u}) - I(T_{\boldsymbol{p}}(\boldsymbol{u}))$, for each position $\boldsymbol{u} \in R$ (by interpolation of I, see Eqn. (24.26)).
   8. Compute the column vector $\boldsymbol{\delta}_{\boldsymbol{p}} = \sum_{\boldsymbol{u} \in R} \boldsymbol{s}_{\boldsymbol{u}}^{\mathsf{T}} \cdot D(\boldsymbol{u})$ (see Eqn. (24.26)).
   9. Calculate the optimal parameter change $\boldsymbol{q}_{\mathrm{opt}} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}}$ (see Eqn. (24.25)).
   10. Update the transformation parameter: $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{q}_{\mathrm{opt}}$ (see Eqn. (24.17)).

   Until $\|\boldsymbol{q}_{\mathrm{opt}}\| < \epsilon$.

[7] The Hessian matrix of a n-variable, real-valued function f is composed of f's second-order partial derivatives (see also Sec. C.2.6 in the Appendix). The Hessian matrix $\mathbf{H}$ is always symmetric.

[8] Moreover, Eqn. (24.31) may be solvable even if the matrix $\bar{\mathbf{H}}$ is almost singular and thus numerically not invertible [160, p. 164].

Alg. 24.1
Lucas-Kanade ("forward-additive") registration algorithm. The origin of the reference image R is placed at its center. The gradient of the image is calculated only once (line 6), but interpolated in every iteration (line 15). Also, the $n \times n$ Hessian matrix $\bar{\mathbf{H}}$ is calculated and inverted in every iteration. The Jacobian of the warp function T is also evaluated repeatedly (line 16), though this is not an expensive calculation, at least for affine warps (lines 32–33). Procedure $\mathsf{Interpolate}(I, \boldsymbol{x}')$ returns the interpolated value of the image I at the continuous position $\boldsymbol{x}' \in \mathbb{R}^2$ (see Ch. 22 for details and possible implementations).

```
1:  LucasKanadeForward($I, R, T, \boldsymbol{p}_{\mathrm{init}}, \epsilon, i_{\max}$)
        Input: $I$, the search image; $R$, the reference image; $T$, a 2D warp
        function that maps any point $\boldsymbol{x} \in \mathbb{R}^2$ to some point $\boldsymbol{x}' = T_{\boldsymbol{p}}(\boldsymbol{x})$,
        with transformation parameters $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$; $\boldsymbol{p}_{\mathrm{init}}$, initial
        estimate of the warp parameters; $\epsilon$, the error limit; $i_{\max}$, the
        maximum number of iterations.
        Returns the modified warp parameter vector $\boldsymbol{p}$ for the best fit
        between $I$ and $R$, or nil if no match could be found.
2:      $(M_R, N_R) \leftarrow$ Size($R$)                       ▷ size of the reference image $R$
3:      $\boldsymbol{x}_{\mathrm{c}} \leftarrow 0.5 \cdot (M_R - 1, N_R - 1)$                 ▷ center of $R$
4:      $\boldsymbol{p} \leftarrow \boldsymbol{p}_{\mathrm{init}}$                      ▷ initial transformation parameters
5:      $n \leftarrow$ Length($\boldsymbol{p}$)                          ▷ parameter count
6:      $(I_{\mathrm{x}}, I_{\mathrm{y}}) \leftarrow$ Gradient($I$)              ▷ calculate the gradient $\nabla I$
7:      $i \leftarrow 0$                                      ▷ iteration counter
8:      do                                            ▷ main loop
9:          $i \leftarrow i + 1$
10:         $\bar{\mathbf{H}} \leftarrow \mathbf{0}_{n,n}$             ▷ $\bar{\mathbf{H}} \in \mathbb{R}^{n \times n}$, initialized to zero
11:         $\boldsymbol{\delta}_{\boldsymbol{p}} \leftarrow \mathbf{0}_n$              ▷ $\boldsymbol{s}_{\boldsymbol{p}} \in \mathbb{R}^n$, initialized to zero
12:         for all positions $\boldsymbol{u} \in (M_R \times N_R)$ do
13:             $\boldsymbol{x} \leftarrow \boldsymbol{u} - \boldsymbol{x}_{\mathrm{c}}$          ▷ position w.r.t. the center of $R$
14:             $\boldsymbol{x}' \leftarrow T_{\boldsymbol{p}}(\boldsymbol{x})$             ▷ warp $\boldsymbol{x}$ to $\boldsymbol{x}'$ by transf. $T_{\boldsymbol{p}}$
                Estimate the gradient of $I$ at the warped position $\boldsymbol{x}'$:
15:             $\nabla \leftarrow \big($Interpolate$(I_{\mathrm{x}}, \boldsymbol{x}')$, Interpolate$(I_{\mathrm{y}}, \boldsymbol{x}')\big)$   ▷ 2D row
                    vector
16:             $\mathbf{J} \leftarrow$ Jacobian($T_{\boldsymbol{p}}, \boldsymbol{x}$)         ▷ Jacobian of $T_{\boldsymbol{p}}$ at pos. $\boldsymbol{x}$
17:             $\boldsymbol{s} \leftarrow (\nabla \cdot \mathbf{J})^{\mathsf{T}}$        ▷ $\boldsymbol{s}$ is a column vector of length $n$
18:             $\mathbf{H} \leftarrow \boldsymbol{s} \cdot \boldsymbol{s}^{\mathsf{T}}$           ▷ outer product, $\mathbf{H}$ is of size $n \times n$
19:             $\bar{\mathbf{H}} \leftarrow \bar{\mathbf{H}} + \mathbf{H}$          ▷ cumulate the Hessian (Eq. 24.30)
20:             $d \leftarrow R(\boldsymbol{u}) -$ Interpolate($I, \boldsymbol{x}'$)     ▷ pixel difference $d \in \mathbb{R}$
21:             $\boldsymbol{\delta}_{\boldsymbol{p}} \leftarrow \boldsymbol{\delta}_{\boldsymbol{p}} + \boldsymbol{s} \cdot d$
22:         $\boldsymbol{q}_{\mathrm{opt}} \leftarrow \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}}$   ▷ Eq. 24.17, or solve $\bar{\mathbf{H}} \cdot \boldsymbol{q}_{\mathrm{opt}} = \boldsymbol{\delta}_{\boldsymbol{p}}$ (Eq. 24.31)
23:         $\boldsymbol{p} \leftarrow \boldsymbol{p} + \boldsymbol{q}_{\mathrm{opt}}$
24:     while $(\lVert \boldsymbol{q}_{\mathrm{opt}} \rVert > \epsilon) \wedge (i < i_{\max})$      ▷ repeat until convergence
25:     if $i < i_{\max}$ then
26:         return $\boldsymbol{p}$
27:     else
28:         return nil

29: Gradient($I$)
        Returns the gradient of $I$ as a pair of maps.
30:     $H_{\mathrm{x}} = \frac{1}{8} \cdot \begin{bmatrix} -1 & 0 & 1 \\ -2 & \mathbf{0} & 2 \\ -1 & 0 & 1 \end{bmatrix}$,   $H_{\mathrm{y}} = \frac{1}{8} \cdot \begin{bmatrix} -1 & -2 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 2 & 1 \end{bmatrix}$
31:     return $(I * H_x, I * H_y)$

32: Jacobian($T_{\boldsymbol{p}}, \boldsymbol{x}$)
        Returns the $2 \times n$ Jacobian matrix of the 2D warp function
        $T_{\boldsymbol{p}}(\boldsymbol{x}) = (T_{\mathrm{x},\boldsymbol{p}}(\boldsymbol{x}), T_{\mathrm{y},\boldsymbol{p}}(\boldsymbol{x}))$ with parameters $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$
        for the spatial position $\boldsymbol{x} \in \mathbb{R}^2$.
33:     return $\begin{pmatrix} \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_0}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_1}(\boldsymbol{x}) & \ldots & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial p_{n-1}}(\boldsymbol{x}) \\ \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_0}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_1}(\boldsymbol{x}) & \ldots & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial p_{n-1}}(\boldsymbol{x}) \end{pmatrix}$   ▷ see Eq. 24.21
```

The complete specification of the Lucas-Kanade algorithm (referred to as the "forward-additive" algorithm in [13]) is given in Alg. 24.1. In addition to the two images I and R, the procedure requires the assumed type of the geometric transformation T, the estimated initial transformation parameters $\boldsymbol{p}_{\mathrm{init}}$, a convergence limit ϵ and the maximum number of iterations $i_{\max}$. The optimal parameter vector $\boldsymbol{p}$ is returned or nil if the optimization did not converge. For better numerical stability, the origin of the reference image R is placed at its center $\boldsymbol{x}_{\mathrm{c}}$ (see line 3), as is also illustrated in Fig. 24.2. The algorithm shows (unlike the just given summary) that it is sufficient to calculate the Jacobian $\mathbf{J}$ (see line 16) and the Hessian matrix $\bar{\mathbf{H}}$ (see line 18) only for the current position ($\boldsymbol{u}$) in the reference image, which implies relatively modest storage requirements. Additional instructions for calculating the Jacobian and Hessian matrices for specific linear transformations T are described in Sec. 24.4. In the case that $\bar{\mathbf{H}}$ cannot be inverted (because it is singular) in line 22, the algorithm could either stop (and return nil) or continue with a small random perturbation of the transformation parameters $\boldsymbol{p}$.

This so-called forward-additive algorithm performs reliably if the assumed type of geometric transformation is correct and the initial parameter estimate is sufficiently close to the actual parameters. However, it is computationally demanding since it requires repeated warping of the gradient image and the Jacobian $\mathbf{J}_{T_{\boldsymbol{p}}}$ as well as the Hessian matrix $\mathbf{H}$ must be re-calculated in each iteration. Very similar results at greatly improved performance are obtained with the "inverse compositional algorithm" described in Sec. 24.3.

24.3 Inverse Compositional Algorithm

This algorithm, described in [14], exchanges the roles of the search image I and the reference image R. As illustrated in Fig. 24.3, the reference image R remains anchored at the original position, while the geometric transformations are applied to (parts of) the search image I. In particular, the transformation $T_{\boldsymbol{p}}$ now describes the mapping from the warped image I' *back* to the original image I. The advantage of this algorithm is that it avoids re-evaluating the Jacobian and Hessian matrices in every iteration while exhibiting convergence properties similar to the Lucas-Kanade (forward-additive) algorithm described in Sec. 24.2.

In this algorithm, the expression to be minimized in each iteration is (cf. Eqn. (24.16))

$$\mathcal{E}(\boldsymbol{q}) = \sum_{\boldsymbol{u}\in R} \bigl[R(T_{\boldsymbol{q}}(\boldsymbol{u})) - I(T_{\boldsymbol{p}}(\boldsymbol{u}))\bigr]^2, \tag{24.32}$$

with respect to the parameter change $\boldsymbol{q}$, producing an optimal change vector $\boldsymbol{q}_{\mathrm{opt}}$. Subsequently, the geometric transformation is updated not by simply *adding* $\boldsymbol{q}_{\mathrm{opt}}$ to the current parameter estimate $\boldsymbol{p}$ (as in Eqn. (24.17)), but by *concatenating* the corresponding warps in the form

$$T_{\boldsymbol{p}'}(\boldsymbol{x}) = (T^{-1}_{\boldsymbol{q}_{\mathrm{opt}}} \circ T_{\boldsymbol{p}})(\boldsymbol{x}) = T_{\boldsymbol{p}}(T^{-1}_{\boldsymbol{q}_{\mathrm{opt}}}(\boldsymbol{x})) \tag{24.33}$$

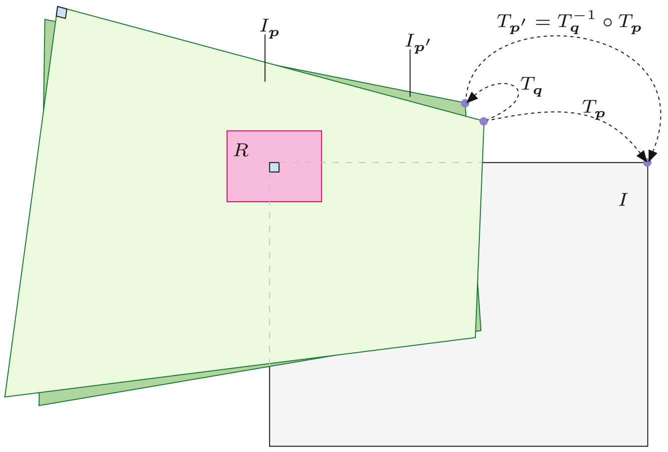

Fig. 24.3 Geometry of the *inverse compositional* registration algorithm. I denotes the search image and R is the reference image. The geometric transformation $T_{\boldsymbol{p}}$ warps the image $I_{\boldsymbol{p}}$ *back* to the original search image I, with $\boldsymbol{p}$ being the initial parameter estimate. Matching is performed between the (unwarped) reference image R and the warped search image $I_{\boldsymbol{p}}$. Note that the reference image R always remains anchored at the origin. In each iteration, the incremental warp $T_{\boldsymbol{q}}$ (with parameter vector $\boldsymbol{q}$) is estimated, mapping the image $I_{\boldsymbol{p}}$ to image $I_{\boldsymbol{p}'}$. The resulting composite warp $T_{\boldsymbol{p}'}$ (mapping $I_{\boldsymbol{p}'}$ back to I) with parameters $\boldsymbol{p}'$ is obtained by concatenating the transformations $T_{\boldsymbol{q}}^{-1}$ and $T_{\boldsymbol{p}}$.

where $\circ$ denotes the concatenation (successive application) of transformations. In the special (but frequent) case of linear geometric transformations, the concatenation is simply accomplished by multiplying the corresponding transformation matrices $\mathbf{M}_{\boldsymbol{p}}$, $\mathbf{M}_{\boldsymbol{q}_{\text{opt}}}$, that is,

$$\mathbf{M}_{\boldsymbol{p}'} = \mathbf{M}_{\boldsymbol{p}} \cdot \mathbf{M}_{\boldsymbol{q}_{\text{opt}}}^{-1} \tag{24.34}$$

(see also Sec. 24.4.4). Also note that the "incremental" transformation $T_{\boldsymbol{q}_{\text{opt}}}$ is *inverted* before it is concatenated with the current warp $T_{\boldsymbol{p}}$, to calculate the parameters of the resulting composite warp $T_{\boldsymbol{p}'}$. Thus the geometric transformation T must be invertible, but this is again no problem with linear (affine or projective) warps.

In summary, given a search image I, a reference image R, a geometric transformation $T_{\boldsymbol{p}}$, an initial parameter estimate $\boldsymbol{p}_{\text{init}}$ and the convergence limit ϵ, the "inverse compositional algorithm" performs the following steps:

A. **Initialize:**
1. Calculate the gradient $\nabla_R(\boldsymbol{x})$ of the reference image R for all $\boldsymbol{x} \in R$.
2. Calculate the Jacobian $\mathbf{J}(\boldsymbol{x}) = \frac{\partial T_{\boldsymbol{p}}}{\partial \boldsymbol{p}}(\boldsymbol{x})$ of the warp function $T_{\boldsymbol{p}}(\boldsymbol{x})$ for all $\boldsymbol{x} \in R$, with $\boldsymbol{p} = \mathbf{0}$.
3. Compute $\boldsymbol{s}_{\boldsymbol{x}} = \nabla_R(\boldsymbol{x}) \cdot \mathbf{J}(\boldsymbol{x})$ for all $\boldsymbol{x} \in R$.
4. Calculate the Hessian matrix as $\mathbf{H} = \sum_R \boldsymbol{s}_{\boldsymbol{x}}^{\mathsf{T}} \cdot \boldsymbol{s}_{\boldsymbol{x}}$ and precalculate its inverse $\mathbf{H}^{-1}$.
5. Initialize the transformation parameters: $\boldsymbol{p} \leftarrow \boldsymbol{p}_{\text{init}}$.

B. **Repeat:**

6. Warp the search image I to I', such that $I'(\boldsymbol{x}) = I(T_{\boldsymbol{p}}(\boldsymbol{x}))$, for all $\boldsymbol{x} \in R$.
7. Compute the (column) vector $\boldsymbol{\delta}_{\boldsymbol{p}} = \sum_R \boldsymbol{s}_{\boldsymbol{x}} \cdot [I'(\boldsymbol{x}) - R(\boldsymbol{x})]$.
8. Estimate the optimal parameter change $\boldsymbol{q}_{\text{opt}} = \mathbf{H}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}}$.
9. Find the warp parameters $\boldsymbol{p}'$, such that $T_{\boldsymbol{p}'} = T_{\boldsymbol{q}_{\text{opt}}}^{-1} \circ T_{\boldsymbol{p}}$.
10. Update the warp parameter $\boldsymbol{p} \leftarrow \boldsymbol{p}'$.

Until $\|\boldsymbol{q}_{\text{opt}}\| < \epsilon$.

```
1:  LucasKanadeInverse($I, R, T, \boldsymbol{p}_{\mathrm{init}}, \epsilon, i_{\max}$)
        Input: $I$, the search image; $R$, the reference image; $T$, a 2D
        warp function that maps any point $\boldsymbol{x} \in \mathbb{R}^2$ to $\boldsymbol{x}' = T_{\boldsymbol{p}}(\boldsymbol{x})$ using
        parameters $\boldsymbol{p} = (p_0, \ldots, p_{n-1})$; $\boldsymbol{p}_{\mathrm{init}}$, initial estimate of the warp
        parameters; $\epsilon$, the error limit (typ. $\epsilon = 10^{-3}$); $i_{\max}$, the maximum
        number of iterations.
        Returns the updated warp parameter vector $\boldsymbol{p}$ for the best fit
        between $I$ and $R$, or nil if no match could be found.

2:      $(M_R, N_R) \leftarrow$ Size($R$)            ▷ size of the reference image $R$
3:      $\boldsymbol{x}_{\mathrm{c}} \leftarrow 0.5 \cdot (M_R - 1, N_R - 1)$            ▷ center of $R$
        Initialize:
4:      $n \leftarrow$ Length($\boldsymbol{p}$)            ▷ parameter count $n$
5:      Create map $\mathsf{S}\colon (M_R \times N_R) \mapsto \mathbb{R}^n$   ▷ $n$ “steepest-descent images”
6:      $(R_{\mathrm{x}}, R_{\mathrm{y}}) \leftarrow$ Gradient($R$)            ▷ $(R_{\mathrm{x}}(\boldsymbol{u}), R_{\mathrm{y}}(\boldsymbol{u}))^{\mathsf{T}} = \nabla_R(\boldsymbol{u})$
7:      $\bar{\mathbf{H}} \leftarrow \mathbf{0}_{n,n}$            ▷ initialize $n \times n$ Hessian matrix to zero
8:      for all positions $\boldsymbol{u} \in (M_R \times N_R)$ do
9:          $\boldsymbol{x} \leftarrow \boldsymbol{u} - \boldsymbol{x}_{\mathrm{c}}$            ▷ centered position
10:         $\nabla_R \leftarrow (R_{\mathrm{x}}(\boldsymbol{u}), R_{\mathrm{y}}(\boldsymbol{u}))$            ▷ 2-dimensional row vector
11:         $\mathbf{J} \leftarrow$ Jacobian($T_{\mathbf{0}}(\boldsymbol{x})$)            ▷ Jacob. of $T$ at pos. $\boldsymbol{x}$ with $\boldsymbol{p} = \mathbf{0}$
12:         $\boldsymbol{s} \leftarrow (\nabla_R \cdot \mathbf{J})^{\mathsf{T}}$            ▷ $\boldsymbol{s}$ is a column vector of length $n$
13:         $\mathsf{S}(\boldsymbol{u}) \leftarrow \boldsymbol{s}$            ▷ keep $\boldsymbol{s}$ for later use
14:         $\mathbf{H} \leftarrow \boldsymbol{s} \cdot \boldsymbol{s}^{\mathsf{T}}$            ▷ outer product, $\mathbf{H}$ is of size $n \times n$
15:         $\bar{\mathbf{H}} \leftarrow \bar{\mathbf{H}} + \mathbf{H}$            ▷ cumulate the Hessian (Eq. 24.30)
16:     $\bar{\mathbf{H}}^{-1} \leftarrow$ Inverse($\bar{\mathbf{H}}$)
17:     if $\bar{\mathbf{H}}^{-1}$ = nil then            ▷ $\bar{\mathbf{H}}$ could not be inverted
18:         return nil            ▷ stop
19:     $\boldsymbol{p} \leftarrow \boldsymbol{p}_{\mathrm{init}}$            ▷ initial parameter estimate
20:     $i \leftarrow 0$            ▷ iteration counter
        Main loop:
21:     do
22:         $i \leftarrow i + 1$
23:         $\boldsymbol{\delta}_{\boldsymbol{p}} \leftarrow \mathbf{0}_n$            ▷ $\boldsymbol{\delta}_{\boldsymbol{p}} \in \mathbb{R}^n$, initialized to zero
24:         for all positions $\boldsymbol{u} \in (M_R \times N_R)$ do
25:             $\boldsymbol{x} \leftarrow \boldsymbol{u} - \boldsymbol{x}_{\mathrm{c}}$            ▷ centered position
26:             $\boldsymbol{x}' \leftarrow T_{\boldsymbol{p}}(\boldsymbol{x})$            ▷ warp $I$ to $I'$
27:             $d \leftarrow$ Interpolate($I, \boldsymbol{x}'$) $- R(\boldsymbol{u})$            ▷ pixel difference $d \in \mathbb{R}$
28:             $\boldsymbol{s} \leftarrow \mathsf{S}(\boldsymbol{u})$            ▷ get pre-calculated $\boldsymbol{s}$
29:             $\boldsymbol{\delta}_{\boldsymbol{p}} \leftarrow \boldsymbol{\delta}_{\boldsymbol{p}} + \boldsymbol{s} \cdot d$
30:         $\boldsymbol{q}_{\mathrm{opt}} \leftarrow \mathbf{H}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}}$            ▷ $\mathbf{H}^{-1}$ is pre-calculated in line 16
31:         $\boldsymbol{p}' \leftarrow$ determine, such that $T_{\boldsymbol{p}'}(\boldsymbol{x}) = T_{\boldsymbol{p}}(T^{-1}_{\boldsymbol{q}_{\mathrm{opt}}}(\boldsymbol{x}))$
32:         $\boldsymbol{p} \leftarrow \boldsymbol{p}'$
33:     while $(\|\boldsymbol{q}_{\mathrm{opt}}\| > \epsilon) \wedge (i < i_{\max})$            ▷ repeat until convergence
34:     return $\begin{cases} \boldsymbol{p} & \text{for } i < i_{\max} \\ \text{nil} & \text{otherwise} \end{cases}$
```

Alg. 24.2
Inverse compositional registration algorithm. The gradient vectors $\nabla_R(u, v)$ of the reference image R are calculated only once (line 6) using procedure Gradient(), as defined in Alg. 24.1. The Jacobian matrix $\mathbf{J}$ of the warp function $T_{\boldsymbol{p}}$ is also evaluated only once (line 11) for $\boldsymbol{p} = \mathbf{0}$ (i.e., the identity mapping) over all positions of the reference image R. Similarly, the Hessian matrix $\mathbf{H}$ and its inverse $\mathbf{H}^{-1}$ are calculated only once (lines 15, 16). $\mathbf{H}^{-1}$ is used to calculate the optimal parameter change vector $\boldsymbol{q}_{\mathrm{opt}}$ in line 30 of the main loop. Procedure Interpolate() in line 27 is the same as in Alg. 24.1. This algorithm is typically about 5–10 times faster than the original Lucas-Kanade (forward) algorithm (see Alg. 24.1), with similar convergence properties.

One can see clearly that in this variant several steps are performed only once at initialization and do not appear inside the main loop. A detailed and concise listing of the inverse compositional algorithm is given in Alg. 24.2 and concrete setups for various linear transformations are described in Sec. 24.4. Since the Jacobian matrix (for the null parameter vector $\boldsymbol{p} = \boldsymbol{0}$) and the Hessian matrix are calculated only once during initialization, this algorithm executes significantly faster than the original Lucas-Kanade (forward-additive) algorithm, while offering similar convergence properties.

24.4 Parameter Setups for Various Linear Transformations

The use of linear transformatons for the geometric mapping T is very common. In the following, we describe detailed setups required for the Lucas-Kanade algorithm for various geometric transformations, such as pure translation as well as affine and projective transformations. This should help to reduce the chance of confusion about the content and structure of the involved vectors and matrices. For additional details and concrete implementations of these transformations readers should consult the associated Java source code in the `imagingbook`[9] library.

24.4.1 Pure Translation

In the case of pure 2D translation, we have $n = 2$ parameters t_{x}, t_{y} and the geometric transformation is (see Eqn. (24.15))

$$\acute{\boldsymbol{x}} = T_{\boldsymbol{p}}(\boldsymbol{x}) = \boldsymbol{x} + \begin{pmatrix} t_{\mathrm{x}} \\ t_{\mathrm{y}} \end{pmatrix}, \tag{24.35}$$

with the parameter vector $\boldsymbol{p} = (p_0, p_1)^{\mathsf{T}} = (t_{\mathrm{x}}, t_{\mathrm{y}})^{\mathsf{T}}$ and $\boldsymbol{x} = (x, y)^{\mathsf{T}}$. Thus the two component functions of the transformation (cf. Eqn. (24.18)) are

$$\begin{aligned} T_{\mathrm{x},\boldsymbol{p}}(\boldsymbol{x}) &= x + t_{\mathrm{x}}, \\ T_{\mathrm{y},\boldsymbol{p}}(\boldsymbol{x}) &= y + t_{\mathrm{y}}, \end{aligned} \tag{24.36}$$

with the 2×2 Jacobian matrix

$$\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{x}}}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{y}}}(\boldsymbol{x}) \\ \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{x}}}(\boldsymbol{x}) & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{y}}}(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{24.37}$$

Note that in this case $\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x})$ is constant,[10] that is, independent of the position $\boldsymbol{x}$ and the parameters $\boldsymbol{p}$. The 2D column vector $\boldsymbol{\delta}_{\boldsymbol{p}}$ (Eqn. (24.26)) is calculated as

$$\boldsymbol{\delta}_{\boldsymbol{p}} = \sum_{\boldsymbol{u} \in R} \bigl[\nabla_I(\underbrace{T_{\boldsymbol{p}}(\boldsymbol{u})}_{\acute{\boldsymbol{u}} \in \mathbb{R}^2}) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\bigr]^{\mathsf{T}} \cdot \underbrace{\bigl[R(\boldsymbol{u}) - I(T_{\boldsymbol{p}}(\boldsymbol{u}))\bigr]}_{D(\boldsymbol{u}) \in \mathbb{R}} \tag{24.38}$$

$$= \sum_{\boldsymbol{u} \in R} \underbrace{\Bigl[\bigl(I_{\mathrm{x}}(\acute{\boldsymbol{u}}), I_{\mathrm{y}}(\acute{\boldsymbol{u}})\bigr) \cdot \bigl(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\bigr)\Bigr]}_{\boldsymbol{s}(\boldsymbol{u}) = (s_0(\boldsymbol{u}), s_1(\boldsymbol{u}))}^{\mathsf{T}} \cdot D(\boldsymbol{u}) = \sum_{\boldsymbol{u} \in R} \begin{pmatrix} I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \end{pmatrix} \cdot D(\boldsymbol{u}) \tag{24.39}$$

$$= \begin{pmatrix} \sum_{\boldsymbol{u}} I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \cdot D(\boldsymbol{u}) \\ \sum_{\boldsymbol{u}} I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \cdot D(\boldsymbol{u}) \end{pmatrix} = \begin{pmatrix} \sum_{\boldsymbol{u}} s_0(\boldsymbol{u}) \cdot D(\boldsymbol{u}) \\ \sum_{\boldsymbol{u}} s_1(\boldsymbol{u}) \cdot D(\boldsymbol{u}) \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \delta_1 \end{pmatrix}, \tag{24.40}$$

[9] Package `imagingbook.pub.geometry.mappings`.

[10] $\mathbf{I}_2$ denotes the 2×2 identity matrix.

where $I_{\mathrm{x}}, I_{\mathrm{y}}$ denote the (estimated) first derivatives of the search image I in x and y-direction, respectively.[11] Thus in this case the *steepest descent images* (Eqn. (24.28)) $s_0(\boldsymbol{x}) = I_{\mathrm{x}}(\acute{\boldsymbol{x}})$ and $s_1(\boldsymbol{x}) = I_{\mathrm{y}}(\acute{\boldsymbol{x}})$ are simply the components of the interpolated gradient of I in the region of the shifted reference image. The associated Hessian matrix (Eqn. (24.29)) is calculated as

$$\bar{\mathbf{H}} = \sum_{\boldsymbol{u}\in R} \big[\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u})) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big]^{\mathsf{T}} \cdot \big[\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u})) \cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big] \tag{24.41}$$

$$= \sum_{\boldsymbol{u}\in R} \big[\underbrace{\nabla_I(\acute{\boldsymbol{u}}) \cdot \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)}_{\boldsymbol{s}(\boldsymbol{u})}\big]^{\mathsf{T}} \cdot \big[\underbrace{\nabla_I(\acute{\boldsymbol{u}}) \cdot \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)}_{\boldsymbol{s}(\boldsymbol{u})}\big] = \sum_{\boldsymbol{u}\in R} \boldsymbol{s}^{\mathsf{T}}(\boldsymbol{u}) \cdot \boldsymbol{s}(\boldsymbol{u}) \tag{24.42}$$

$$= \sum_{\boldsymbol{u}\in R} \nabla_I^{\mathsf{T}}(\acute{\boldsymbol{u}}) \cdot \nabla_I(\acute{\boldsymbol{u}}) = \sum_{\boldsymbol{u}\in R} \begin{pmatrix} I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \end{pmatrix} \cdot \big(I_{\mathrm{x}}(\acute{\boldsymbol{u}}),\, I_{\mathrm{y}}(\acute{\boldsymbol{u}})\big) \tag{24.43}$$

$$= \sum_{\boldsymbol{u}\in R} \begin{pmatrix} I_{\mathrm{x}}^2(\acute{\boldsymbol{u}}) & I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot I_{\mathrm{y}}(\acute{\boldsymbol{u}}) & I_{\mathrm{y}}^2(\acute{\boldsymbol{u}}) \end{pmatrix} \tag{24.44}$$

$$= \begin{pmatrix} \Sigma\, I_{\mathrm{x}}^2(\acute{\boldsymbol{u}}) & \Sigma\, I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \\ \Sigma\, I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot I_{\mathrm{y}}(\acute{\boldsymbol{u}}) & \Sigma\, I_{\mathrm{y}}^2(\acute{\boldsymbol{u}}) \end{pmatrix} = \begin{pmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{pmatrix}, \tag{24.45}$$

again with $\acute{\boldsymbol{u}} = T_{\boldsymbol{p}}(\boldsymbol{u})$. Since $\bar{\mathbf{H}}$ is symmetric ($H_{01} = H_{10}$) and only of size 2×2, its *inverse* can be easily obtained in closed form:

$$\bar{\mathbf{H}}^{-1} = \frac{1}{H_{00} \cdot H_{11} - H_{01} \cdot H_{10}} \cdot \begin{pmatrix} H_{11} & -H_{01} \\ -H_{10} & H_{00} \end{pmatrix} \tag{24.46}$$

$$= \frac{1}{H_{00} \cdot H_{11} - H_{01}^2} \cdot \begin{pmatrix} H_{11} & -H_{01} \\ -H_{01} & H_{00} \end{pmatrix}. \tag{24.47}$$

The resulting optimal parameter increment (see Eqn. (24.25)) is

$$\boldsymbol{q}_{\mathrm{opt}} = \begin{pmatrix} t'_x \\ t'_y \end{pmatrix} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}} = \bar{\mathbf{H}}^{-1} \cdot \begin{pmatrix} \delta_0 \\ \delta_1 \end{pmatrix} \tag{24.48}$$

$$= \frac{1}{H_{11}\cdot H_{22} - H_{12}^2} \cdot \begin{pmatrix} H_{11} \cdot \delta_0 - H_{01} \cdot \delta_1 \\ H_{00} \cdot \delta_1 - H_{01} \cdot \delta_0 \end{pmatrix}, \tag{24.49}$$

with δ_0, δ_1 as defined in Eqn. (24.40). Alternatively the same result could be obtained by solving Eqn. (24.31) for $\boldsymbol{q}_{\mathrm{opt}}$.

24.4.2 Affine Transformation

An affine transformation in 2D can be expressed (for example) with homogeneous coordinates[12] in the form

$$T_{\boldsymbol{p}}(\boldsymbol{x}) = \begin{pmatrix} 1+a & b & t_{\mathrm{x}} \\ c & 1+d & t_{\mathrm{y}} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{24.50}$$

with $n = 6$ parameters $\boldsymbol{p} = (p_0, \ldots, p_5)^{\mathsf{T}} = (a, b, c, d, t_{\mathrm{x}}, t_{\mathrm{y}})^{\mathsf{T}}$. This parameterization of the affine transformation implies that the *null*

[11] See Sec. C.3.1 in the Appendix for how to estimate gradients of discrete images.

[12] See also Chapter 21, Secs. 21.1.2 and 21.1.3.

parameter vector ($\boldsymbol{p} = \boldsymbol{0}$) corresponds to the *identity* transformation. The component functions of this transformation thus are

$$\begin{aligned} T_{\mathrm{x},\boldsymbol{p}}(\boldsymbol{x}) &= (1+a)\cdot x + b\cdot y + t_{\mathrm{x}}, \\ T_{\mathrm{y},\boldsymbol{p}}(\boldsymbol{x}) &= c\cdot x + (1+d)\cdot y + t_{\mathrm{y}}, \end{aligned} \tag{24.51}$$

and the associated Jacobian matrix at some position $\boldsymbol{x} = (x, y)$ is

$$\mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial a} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial b} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial c} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial d} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{x}}} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{y}}} \\ \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial a} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial b} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial c} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial d} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{x}}} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{y}}} \end{pmatrix}(\boldsymbol{x}) \tag{24.52}$$

$$= \begin{pmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{pmatrix}. \tag{24.53}$$

Note that in this case the Jacobian only depends on the position $\boldsymbol{x} = (x, y)$, not on the transformation parameters $\boldsymbol{p}$. It can thus be pre-calculated once for all positions $\boldsymbol{x}$ of the reference image R. The 6-dimensional column vector $\boldsymbol{\delta_p}$ (Eqn. (24.26)) is obtained as

$$\boldsymbol{\delta_p} = \sum_{\boldsymbol{u}\in R} \underbrace{\big[\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u}))\cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big]}_{\boldsymbol{s}(\boldsymbol{u})}^{\mathsf{T}} \cdot \underbrace{\big[R(\boldsymbol{u}) - I(T_{\boldsymbol{p}}(\boldsymbol{u}))\big]}_{D(\boldsymbol{u})} \tag{24.54}$$

$$= \sum_{\boldsymbol{u}\in R} \left[(I_{\mathrm{x}}(\acute{\boldsymbol{u}}), I_{\mathrm{y}}(\acute{\boldsymbol{u}}))\cdot \begin{pmatrix} x & y & 0 & 0 & 1 & 0 \\ 0 & 0 & x & y & 0 & 1 \end{pmatrix} \right]^{\mathsf{T}} \cdot D(\boldsymbol{u}) \tag{24.55}$$

$$= \sum_{\boldsymbol{u}\in R} \begin{pmatrix} I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \end{pmatrix} \cdot D(\boldsymbol{u}) \;=\; \sum_{\boldsymbol{u}\in R} \begin{pmatrix} s_0(\boldsymbol{u}) \\ s_1(\boldsymbol{u}) \\ s_2(\boldsymbol{u}) \\ s_3(\boldsymbol{u}) \\ s_4(\boldsymbol{u}) \\ s_5(\boldsymbol{u}) \end{pmatrix} \cdot D(\boldsymbol{u}) \tag{24.56}$$

$$= \sum_{\boldsymbol{u}\in R} \begin{pmatrix} s_0(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ s_1(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ s_2(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ s_3(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ s_4(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ s_5(\boldsymbol{u})\cdot D(\boldsymbol{u}) \end{pmatrix} = \begin{pmatrix} \Sigma\, s_0(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ \Sigma\, s_1(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ \Sigma\, s_2(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ \Sigma\, s_3(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ \Sigma\, s_4(\boldsymbol{u})\cdot D(\boldsymbol{u}) \\ \Sigma\, s_5(\boldsymbol{u})\cdot D(\boldsymbol{u}) \end{pmatrix}, \tag{24.57}$$

again with $\acute{\boldsymbol{u}} = T_{\boldsymbol{p}}(\boldsymbol{u})$. The corresponding Hessian matrix (of size 6×6) is found as

$$\bar{\mathbf{H}} = \sum_{\boldsymbol{u}\in R} \big[\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u}))\cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big]^{\mathsf{T}} \cdot \big[\nabla_I(T_{\boldsymbol{p}}(\boldsymbol{u}))\cdot \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{u})\big] \tag{24.58}$$

$$= \sum_{\boldsymbol{x}\in R} \boldsymbol{s}^{\mathsf{T}}(\boldsymbol{u})\cdot \boldsymbol{s}(\boldsymbol{u}) = \sum_{\boldsymbol{x}\in R} \begin{pmatrix} I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \end{pmatrix}^{\mathsf{T}} \cdot \begin{pmatrix} I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot x \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}})\cdot y \\ I_{\mathrm{x}}(\acute{\boldsymbol{u}}) \\ I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \end{pmatrix} = \tag{24.59}$$

$$\begin{pmatrix}
\Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})x^2 & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x^2 & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x \\
\Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})y^2 & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y^2 & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y \\
\Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x^2 & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})x^2 & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})x \\
\Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y^2 & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})xy & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})y^2(\acute{\boldsymbol{u}}) & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})y \\
\Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{x}}^2(\acute{\boldsymbol{u}}) & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}}) \\
\Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})x & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})y & \Sigma I_{\mathrm{x}}(\acute{\boldsymbol{u}})I_{\mathrm{y}}(\acute{\boldsymbol{u}}) & \Sigma I_{\mathrm{y}}^2(\acute{\boldsymbol{u}})
\end{pmatrix}. \tag{24.60}$$

Finally, the optimal parameter increment (see Eqn. (24.25)) is calculated as

$$\boldsymbol{q}_{\text{opt}} = \left(a', b', c', d', t'_x, t'_y\right)^{\mathsf{T}} = \bar{\mathbf{H}}^{-1} \cdot \boldsymbol{\delta}_{\boldsymbol{p}} \tag{24.61}$$

or, equivalently, by solving $\mathbf{H} \cdot \boldsymbol{q}_{\text{opt}} = \boldsymbol{\delta}_{\boldsymbol{p}}$ (see Eqn. (24.31)). For both approaches, no closed-form solution is possible but numerical methods must be used.

24.4.3 Projective Transformation

A projective transformation[13] can be expressed (for example) with homogeneous coordinates in the form

$$T_{\boldsymbol{p}}(\boldsymbol{x}) = \mathbf{M}_{\boldsymbol{p}} \cdot \boldsymbol{x} = \begin{pmatrix} 1+a & b & t_{\mathrm{x}} \\ c & 1+d & t_{\mathrm{y}} \\ e & f & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}, \tag{24.62}$$

with $n = 8$ parameters $\boldsymbol{p} = (p_0, \ldots, p_7) = (a, b, c, d, e, f, t_{\mathrm{x}}, t_{\mathrm{y}})$. Again the null parameter vector corresponds to the identity transformation. In this case, the results need to be converted back to non-homogeneous coordinates (see Ch. 21, Sec. 21.1.2), which yields the transformation's effective (nonlinear) component functions

$$T_{\mathrm{x},\boldsymbol{p}}(\boldsymbol{x}) = \frac{(1+a)\cdot x + b \cdot y + t_{\mathrm{x}}}{e \cdot x + f \cdot y + 1} = \frac{\alpha}{\gamma}, \tag{24.63}$$

$$T_{\mathrm{y},\boldsymbol{p}}(\boldsymbol{x}) = \frac{c \cdot x + (1+d)\cdot y + t_{\mathrm{y}}}{e \cdot x + f \cdot y + 1} = \frac{\beta}{\gamma}, \tag{24.64}$$

with $\boldsymbol{x} = (x, y)$ and

$$\alpha = (1+a)\cdot x + b \cdot y + t_{\mathrm{x}}, \tag{24.65}$$

$$\beta = c \cdot x + (1+d)\cdot y + t_{\mathrm{y}}, \tag{24.66}$$

$$\gamma = e \cdot x + f \cdot y + 1. \tag{24.67}$$

In this case, the associated Jacobian matrix for position $\boldsymbol{x} = (x, y)$,

$$\begin{aligned} \mathbf{J}_{T_{\boldsymbol{p}}}(\boldsymbol{x}) &= \begin{pmatrix} \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial a} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial b} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial c} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial d} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial e} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial f} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{x}}} & \frac{\partial T_{\mathrm{x},\boldsymbol{p}}}{\partial t_{\mathrm{y}}} \\ \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial a} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial b} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial c} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial d} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial e} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial f} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{x}}} & \frac{\partial T_{\mathrm{y},\boldsymbol{p}}}{\partial t_{\mathrm{y}}} \end{pmatrix}(\boldsymbol{x}) \\ &= \frac{1}{\gamma} \cdot \begin{pmatrix} x & y & 0 & 0 & -\frac{x\cdot\alpha}{\gamma} & -\frac{y\cdot\alpha}{\gamma} & 1 & 0 \\ 0 & 0 & x & y & -\frac{x\cdot\beta}{\gamma} & -\frac{y\cdot\beta}{\gamma} & 0 & 1 \end{pmatrix}, \end{aligned} \tag{24.68}$$

depends on both the position $\boldsymbol{x}$ as well as the transformation parameters $\boldsymbol{p}$. The setup for the resulting Hessian matrix $\mathbf{H}$ is analogous to Eqns. (24.58)–(24.61).

24.4.4 Concatenating Linear Transformations

The "inverse compositional" algorithm described in Sec. 24.3 requires the concatenation of geometric transformations (see Eqn. (24.33)). In

[13] See also Chapter 21, Sec. 21.1.4.

particular, if $T_{\boldsymbol{p}}, T_{\boldsymbol{q}}$ are *linear* transformations (in homogeneous coordinates, see Eqn. (24.62)), with associated transformation matrices $\mathbf{M}_{\boldsymbol{p}}$ and $\mathbf{M}_{\boldsymbol{q}}$ (such that $T_{\boldsymbol{p}}(\boldsymbol{x}) = \mathbf{M}_{\boldsymbol{p}} \cdot \boldsymbol{x}$ and $T_{\boldsymbol{q}}(\boldsymbol{x}) = \mathbf{M}_{\boldsymbol{q}} \cdot \boldsymbol{x}$, respectively), the matrix for the concatenated transformation,

$$T_{\boldsymbol{p}'}(\boldsymbol{x}) = (T_{\boldsymbol{p}} \circ T_{\boldsymbol{q}})(\boldsymbol{x}) = T_{\boldsymbol{q}}(T_{\boldsymbol{p}}(\boldsymbol{x})) \tag{24.69}$$

is simply the product of the original matrices, that is,

$$\mathbf{M}_{\boldsymbol{p}'} \cdot \boldsymbol{x} = \mathbf{M}_{\boldsymbol{q}} \cdot \mathbf{M}_{\boldsymbol{p}} \cdot \boldsymbol{x}. \tag{24.70}$$

The resulting parameter vector $\boldsymbol{p}'$ for the composite transformation $T_{\boldsymbol{p}'}$ can be simply extracted from the corresponding elements of the matrix $\mathbf{M}_{\boldsymbol{p}'}$ (see Eqn. (24.50) and Eqn. (24.62)), respectively.

24.5 Example

Figure 24.4 shows an example for using the classic Lucas-Kanade (forward-additive) matcher. Initially, a rectangular region Q is selected in the search image I, marked by the green rectangle in Fig. 24.4(a,b), which specifies the approximate position of the reference image. To create the (synthetic) reference image R, all four corners of the rectangle Q were perturbed randomly in x- and y-direction by Gaussian noise (with $\sigma = 2.5$) in x- and y-direction. The resulting quadrilateral Q' (red outline in Fig. 24.4(a,b)) specifies the region in image I where the reference image R was extracted by transformation and interpolation (see Fig. 24.4(d)). The matching process starts from the rectangle Q, which specifies the *initial* warp transformation T_{init}, given by the green rectangle (Q), while the real (but unknown) transformation corresponds to the red quadrilateral (Q'). Each iteration of the matcher updates the warp transformation T. The blue circles in Fig. 24.4(b) mark the corners of the back-projected reference frame under the changing transformation T; the radius of the circles corresponds to the remaining registration error between the reference image R and the current subimage of I.

Figure 24.4(e) shows the steepest-descent images $s_0, \ldots, s_7$ (see Eqn. (24.28)) for the first iteration. Each of these images is of the same size as R and corresponds to one of the 8 parameters $a, b, c, d, e, f, t_{\text{x}}, t_{\text{y}}$ of the projective warp transformation (see Eqn. (24.62)). The value $s_k(u, v)$ in a particular image s_k corresponds to the optimal change of the transformation parameter k with respect to the associated image position (u, v). The actual change of parameter k is calculated by averaging over all positions (u, v) of the reference image R.

The example demonstrates the robustness and fast convergence of the classic Lucas-Kanade matcher, which typically requires only 5–20 iterations. In this case, the matcher performed 7 iterations to converge (with convergence limit $\epsilon = 0.00001$). In comparison, the inverse-compositional matcher typically requires more iterations and is less tolerant to deviations of the initial warp transformation,

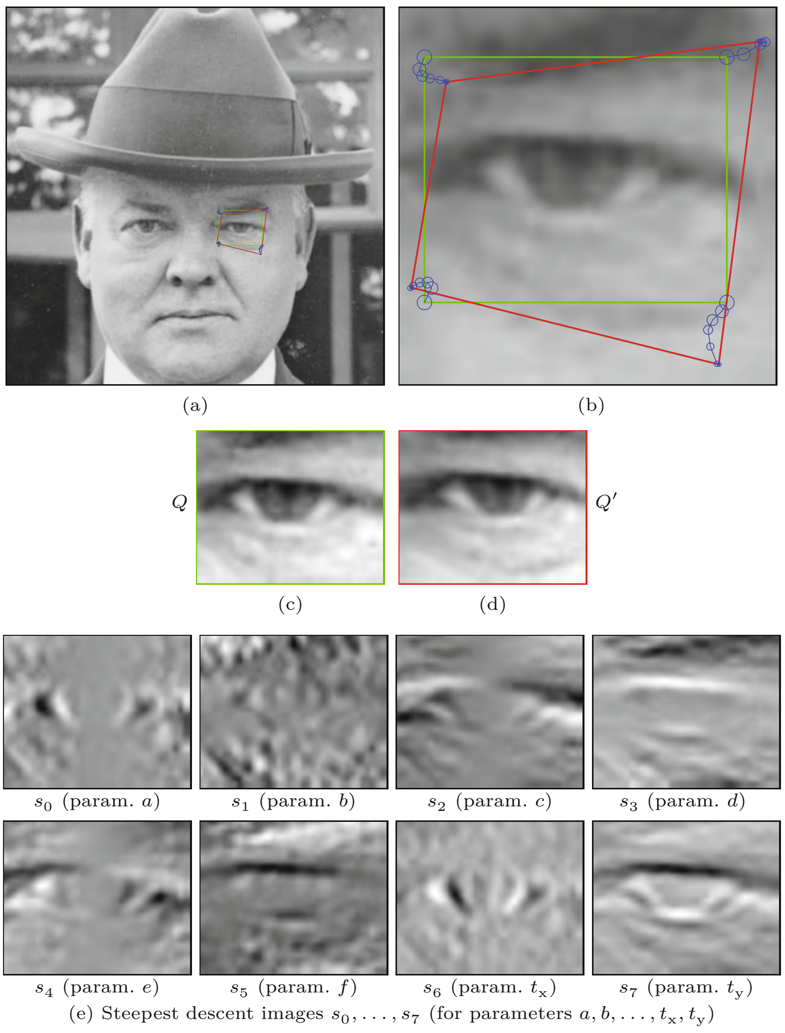

Fig. 24.4
Lucas-Kanade (forward-additive) matcher with projective warp transformation. Original image I (a b); the initial warp transformation T_{init} is visualized by the green rectangle Q, which corresponds to the subimage shown in (c). The actual reference image R (d) has been extracted from the red quadrilateral Q' (by transformation and interpolation). The blue circles mark the corners of the back-projected reference image under the changing transformation $T_{\boldsymbol{p}}$. The radius of each circle corresponds to the registration error between the transformed reference image R and the currently overlapping part of the search image I. The *steepest-descent images* $s_0, \ldots, s_7$ (one for each of the 8 parameters $a, b, c, d, e, f, t_x, t_y$ of the projective transformation) for the first iteration are shown in (e). These images are of the same size as the reference image R.

that is, has a smaller convergence range than the additive-forward algorithm.[14]

24.6 Java Implementation

The algorithms described in this chapter have been implemented in Java, with the source code available as part of the `imagingbook`[15] library on the book's accompanying website. As usual, most Java variables and methods in the online code have been named similarly to the identifiers used in the text for easier understanding.

[14] In fact, the inverse-compositional algorithm does not converge with this particular example.

[15] Package `imagingbook.pub.lucaskanade`.

`LucasKanadeMatcher` (class)

This is the (abstract) super-class of the concrete matchers (`ForwardAdditiveMatcher`, `InverseCompositionalMatcher`) described further. It defines a static inner class `Parameters`[16] with public parameter fields such as

`tolerance` ($= \epsilon$, default 0.00001),
`maxIterations` ($= i_{\max}$, default 100).

In addition, class `LucasKanadeMatcher` itself provides the following public methods:

`LinearMapping getMatch (ProjectiveMapping T)`
Performs a complete match on the given image pair `I`, `R` (required by the sub-class constructors), with `T` used as the initial geometric transformation. The transformation object `T` may be of any subtype of `ProjectiveMapping`,[17] including `Translation` and `AffineMapping`. The method returns a new transformation object for the optimal match, or `null` if the matcher did not converge.

`ProjectiveMapping iterateOnce (ProjectiveMapping T)`
This method performs a single matching iteration with the current warp transformation `T`. It is typically invoked repeatedly after an initial call to `initializeMatch()`. The updated warp transformation is returned, or `null` if the iteration was unsuccessful (e.g., if the Hessian matrix could not be inverted).

`boolean hasConverged ()`
Returns `true` if (and only if) the minimization criteria (specified by the `tolerance` parameter) have been reached. This method is typically used to terminate the optimization loop after calling `iterateOnce()`.

`Point2D[] getReferencePoints ()`
Returns the four corner points of the bounding rectangle of the reference image R, centered at the origin. All warp transformations (including `Tinit` and `Tp`) refer to these coordinates. Note that the returned point coordinates are generally non-integer values; for example, for a reference image size 11×8, the reference corner points are $A = (-5, -3.5)$, $B = (5, -3.5)$, $C = (5, 3.5)$, and $D = (-5, 3.5)$ (see Fig. 24.5).

`ProjectiveMapping getReferenceMappingTo (Point2D[] Q)`
Calculates the (linear) geometric transformation between the reference image `R` (centered at the origin) and the quadrilateral specified by the point sequence `Q`. The type of the returned mapping depends on the number of points in `Q` (max. 4).

`double getRmsError ()`
Returns the RMS error between images `I` and `R` for the most recent iteration (usually called after `iterateOnce()`).

[16] See the usage example in Prog. 24.1.
[17] Class `ProjectiveMapping` is described in Chapter 21, Sec. 21.1.4.

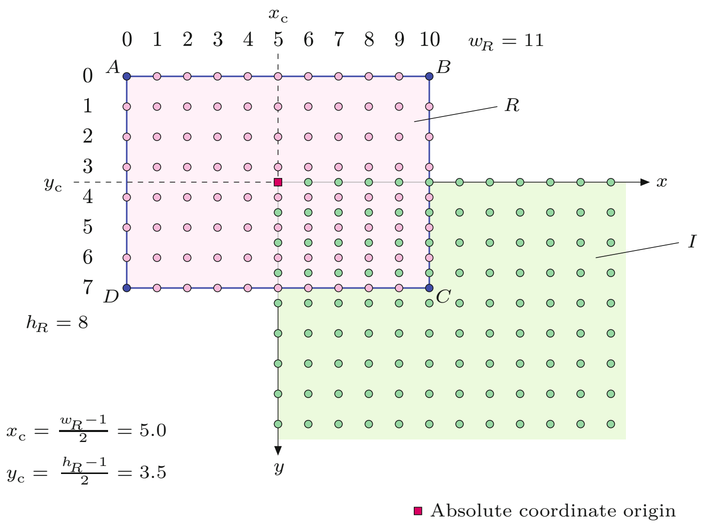

Fig. 24.5
Reference coordinates. The center of the reference image R is aligned with the origin of the search image I (red square), which is taken as the absolute origin. Image samples (indicated by round dots) are assumed to be located at integer positions. In this example, the reference image R is of size $w_R = 11$ and $h_R = 8$, thus the center coordinates are $x_c = 5.0$ and $y_c = 3.5$. In the x/y coordinate frame of I (i.e., absolute coordinates), the four corners of R's bounding rectangle are $A = (-5, -3.5)$, $B = (5, -3.5)$, $C = (5, 3.5)$ and $D = (-5, 3.5)$. All warp transformations refer to these reference points (cf. Figs. 24.2 and 24.3).

`LucasKanadeForwardMatcher` (class)

This sub-class of `LucasKanadeMatcher` implements the Lucas-Kanade ("forward-additive") algorithm, as outlined in Alg. 24.1. It provides the aforementioned methods for `LucasKanadeMatcher` and two constructors:

```
LucasKanadeForwardMatcher (FloatProcessor I,
        FloatProcessor R)
```

Here `I` is the search image, `R` is the (smaller) reference image. It creates a new instance of `LucasKanadeForwardMatcher` using default parameter values.

```
LucasKanadeForwardMatcher (FloatProcessor I,
        FloatProcessor R, Parameters params)
```

Creates a new instance of type `LucasKanadeForwardMatcher` using the specific settings in `params`.

`LucasKanadeInverseMatcher` (class)

This sub-class of `LucasKanadeMatcher` implements the "inverse compositional" algorithm, as described in Alg. 24.2. It provides the same methods and constructors as class `LucasKanadeForwardMatcher`:

```
LucasKanadeInverseMatcher (FloatProcessor I,
        FloatProcessor R).
LucasKanadeInverseMatcher (FloatProcessor I,
        FloatProcessor R, Parameters params).
```

24.6.1 Application Example

The code example in Prog. 24.1 demonstrates the use of the Lucas-Kanade API. The ImageJ plugin is applied to the search image I (the current image) and requires a rectangular ROI to be selected, which is taken as the initial guess for the match region. The reference image is created synthetically by extracting a warped sub-image

Prog. 24.1
Lucas-Kanade code example (ImageJ plugin). This plugin is applied to the search image (I) and assumes that a rectangular ROI is selected whose bounding rectangle and corner points (`Q`) are obtained in lines 22–27. The search image `I` is copied from the current image (as a `FloatProcessor` object) in line 19. The size of the reference image `R` (created in line 24) is defined by the ROI rectangle, whose corner points `Q` also determine the initial parameters of the geometric transformation `Tinit` (line 27 and 37, respectively). The synthetic reference image `R` (with the same size as the ROI) is extracted from the search image by warping from a quadrilateral (`QQ`), which is obtained by randomly perturbing the corner points of the selected ROI (lines 28–29). A new matcher object is created in lines 32–33, in this case of type `LucasKanadeForwardMatcher` (alternatively, `LucasKanadeInverseMatcher` could have been used). The actual match operation is performed in lines 40–44. It consists of a simple `do-while` loop which is terminated if either, the transformation `T` becomes invalid (`null`), the matcher has converged or the maximum number of iterations has been reached. Alternatively, lines 40–44 could have been replaced by the statement `T = matcher.getMatch(Tinit)`. If the matcher has converged, the final transformation `Tp` maps to the best-matching sub-image of `I`.

```
 1 import ...
 2
 3 public class LucasKanade_Demo implements PlugInFilter {
 4
 5   static int maxIterations = 100;
 6
 7   public int setup(String args, ImagePlus img) {
 8     return DOES_8G + ROI_REQUIRED;
 9   }
10
11   public void run(ImageProcessor ip) {
12     Roi roi = img.getRoi();
13     if (roi != null && roi.getType() != Roi.RECTANGLE) {
14       IJ.error("Rectangular selection required!)");
15       return;
16     }
17
18     // Step 1: create the search image I:
19     FloatProcessor I = ip.convertToFloatProcessor();
20
21     // Step 2: create the (empty) reference image R:
22     Rectangle roiR = roi.getBounds();
23     FloatProcessor R =
24         new FloatProcessor(roiR.width, roiR.height);
25
26     // Step 3: perturb the rectangle Q to Q' to extract reference image R:
27     Point2D[] Q  = getCornerPoints(roiR); // = Q
28     Point2D[] QQ = perturbGaussian(Q); // = Q'
29     (new ImageExtractor(I)).extractImage(R, QQ);
30
31     // Step 4: create the Lucas-Kanade matcher (forward or inverse):
32     LucasKanadeMatcher matcher =
33         new LucasKanadeForwardMatcher(I, R);
34
35     // Step 5: calculate the initial mapping Tinit:
36     ProjectiveMapping Tinit =
37         matcher.getReferenceMappingTo(Q);
38
39     // Step 6: initialize and run the matching loop:
40     ProjectiveMapping T = Tinit;
41     do {
42       T = matcher.iterateOnce(T);
43     } while (T != null && !matcher.hasConverged() &&
44           matcher.getIteration() < maxIterations);
45
46     // Step 7: evaluate the result:
47     if (T == null || !matcher.hasConverged()) {
48       IJ.log("no match found!");
49       return;
50     }
51     else {
52       ProjectiveMapping Tfinal = T;
53       ...
54   }
55
56 }
```

of I from a random quadrilateral around the selected ROI.[18] The required geometric transformations (such as `ProjectiveMapping`, `AffineMapping`, `Translation` etc.) are described in Chapter 21, Sec. 21.1.

The example demonstrates how the Lucas-Kanade matcher is initialized and called repeatedly inside the optimization loop using a projective transformation. This usage mode is specifically intended for testing purposes, since it allows to retrieve the state of the matcher after every iteration. The same result could be obtained by replacing the whole loop (lines 40–44 in Prog. 24.1) with the single instruction

```
    ProjectiveMapping T = matcher.getMatch(Tinit);
```

Moreover, in line 33, the `LucasKanadeForwardMatcher` could be replaced by an instance of `LucasKanadeInverseMatcher` without any additional changes. For further details, see the complete source code on the book's website.

24.7 Exercises

Exercise 24.1. Determine the general structure of the Hessian matrix for the projective transformation (see Sec. 24.4.3), analogous to the affine transformation in Eqns. (24.58)–(24.60).

Exercise 24.2. Create comparative statistics of the convergence properties of the classes `ForwardAdditiveMatcher` and `InverseCompositionalMatcher` by evaluating the number of iterations required including the percentage of failures. Use a test scenario with randomly perturbed reference regions as shown in Prog. 24.1.

Exercise 24.3. It is sometimes suggested to refine the warp transformation step-by-step instead of using the full transformation for the whole matching process. For example, one could first match with a pure translation model, then—starting from the result of the first match—switch to an affine transformation model, and eventually apply a full projective transformation. Explore this idea and find out whether this can yield a more robust matching process.

Exercise 24.4. Adapt the 2D Lucas-Kanade method described in Sec. 24.2 for the registration of discrete *1D* signals under shifting and scaling. Given is a search signal $I(u)$, for $u = 0, \ldots, M_I - 1$, and a reference signal $R(u)$, for $u = 0, \ldots, M_R - 1$. It is assumed that I contains a transformed version of R, which is specified by the mapping $T_{\boldsymbol{p}}(x) = s \cdot x + t$, with the two unknown parameters $\boldsymbol{p} = (s, t)$. A practical application could be the registration of neighboring image lines under perspective distortion.

Exercise 24.5. Use the Lucas-Kanade matcher to design a tracker that follows a given reference patch through a sequence of N images. Hint: In ImageJ, an image sequence (AVI-video or multi-frame TIFF)

[18] The class `ImageExtractor`, used to extract the warped sub-image, is part of the `imagingbook` library (package `imagingbook.lib.image`).

can be imported as an `ImageStack` and simply processed frame-by-frame. Select the original reference patch in the first frame of the image sequence and use its position to calculate the initial warp transformation to find a match in the second image. Subsequently, take the match obtained in the second image as the initial transformation for the third image, etc. Consider two approaches: (a) use the initial patch as the reference image for *all* frames of the sequence or (b) extract a new reference image for each pair of frames.