Intelligent Control Systems

Visual Tracking (1)

— Direct Pixel-Intensity-based Methods —

Shingo Kagami
Graduate School of Information Sciences,
Tohoku University
swk(at)ic.is.tohoku.ac.jp

http://www.ic.is.tohoku.ac.jp/ja/swk/

Sample codes for this week

- Open https://github.com/shingo-kagami/ic.git
- Click the green button “Code” and click “Download Zip”
- Copy the files whose names start from ic03*** to C:¥ic2022¥sample

If you are a Git user, you may simply run:

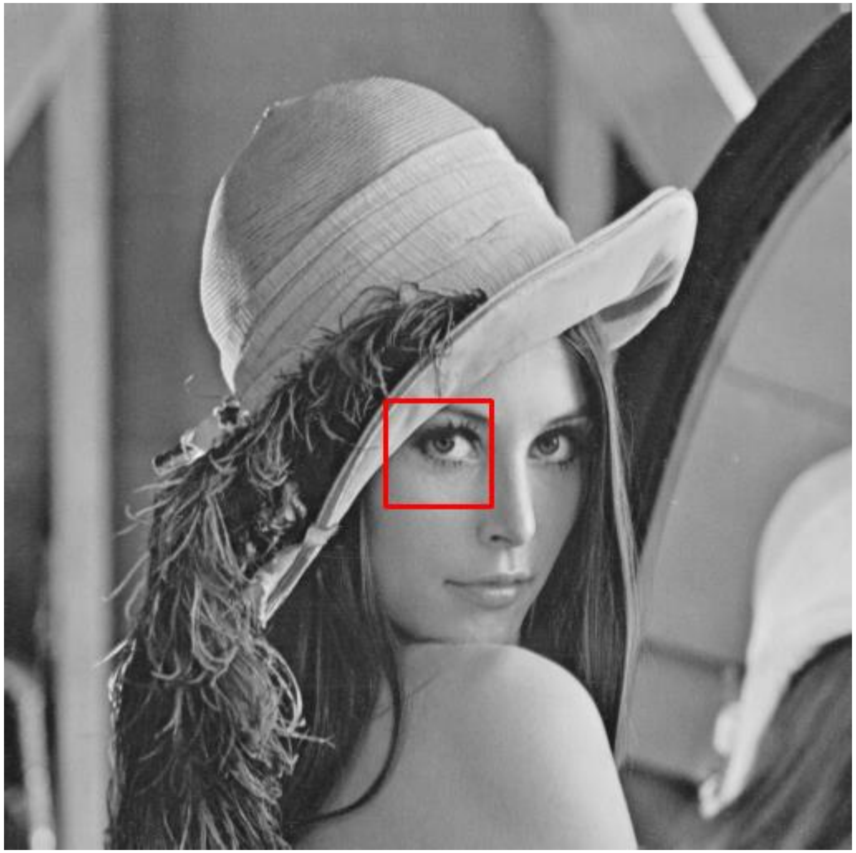
```
cd C:¥ic2022¥sample
git pull
```

Agenda

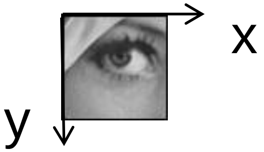
- Template Matching by Brute-force Search
- Template Matching by Gradient-based Search
- Feature Point Detection
- Gradient-based Search for General Warps

Visual Tracking

input image



template image $T_{x,y}$



Matching Problem:

- To find the area with the best similarity to the template

How?

- by evaluating a similarity measure or a dissimilarity measure for every possible position

Matching is often called "tracking" when it is sequentially done with time

Detection vs Tracking

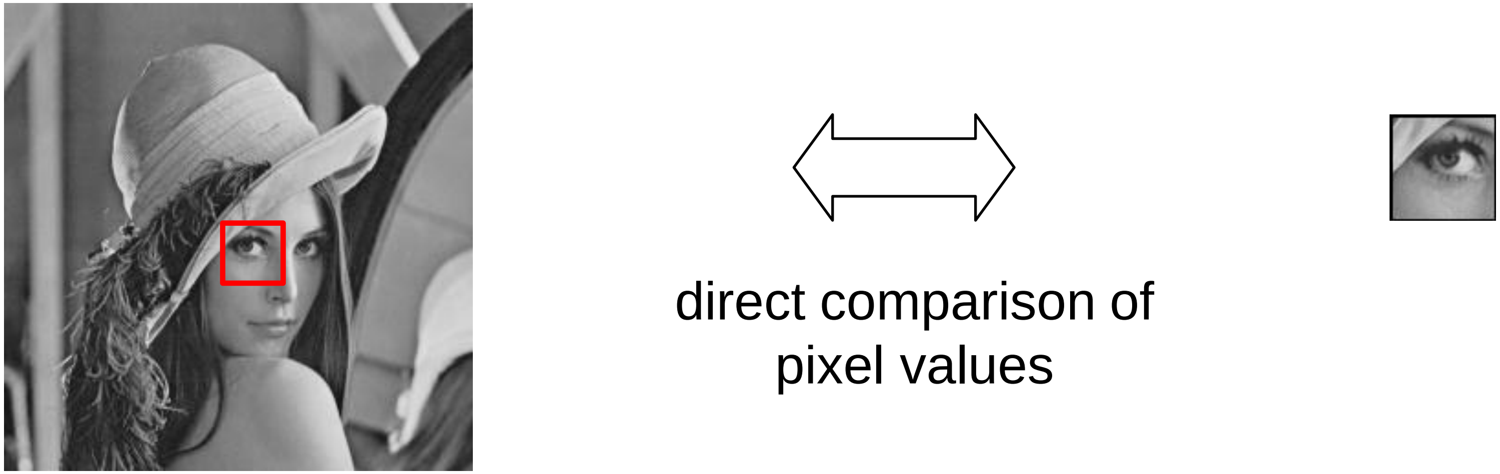
Matching problem is called *detection* when:

Target object is found out of the entire image without relying on knowledge in previous frames

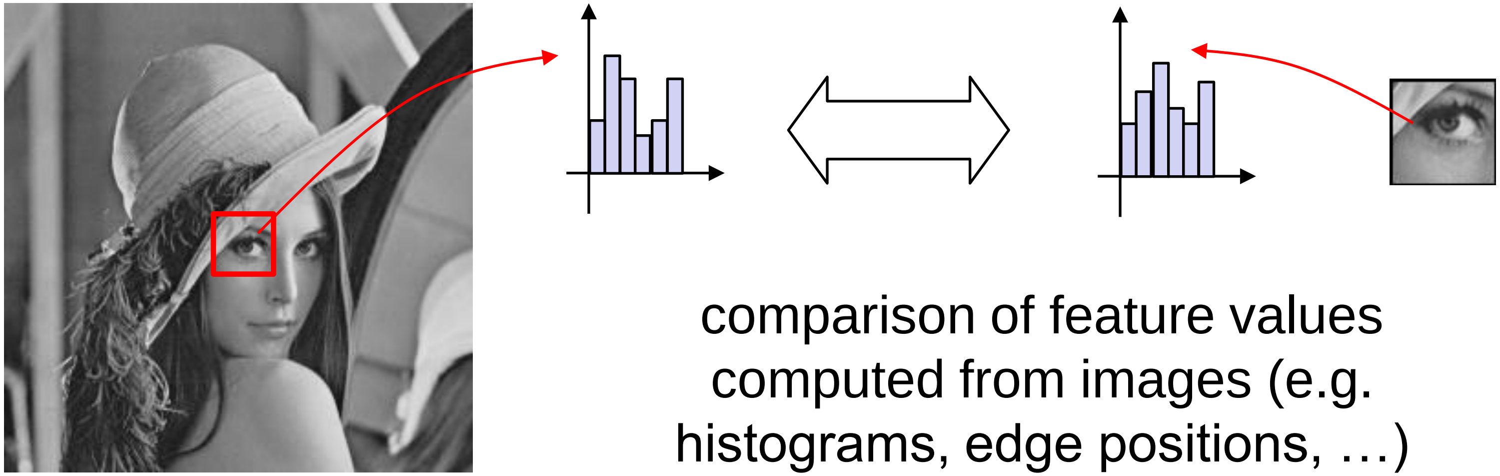
- If we detect the target object every frame, it can be regarded as a kind of tracking (Tracking by Detection)
- However, detection is usually computationally demanding

Hence, when real-time tracking is needed, we usually try to utilize our knowledge in previous frames; once failed, we fall back to detection

Feature-based Methods vs Direct Methods

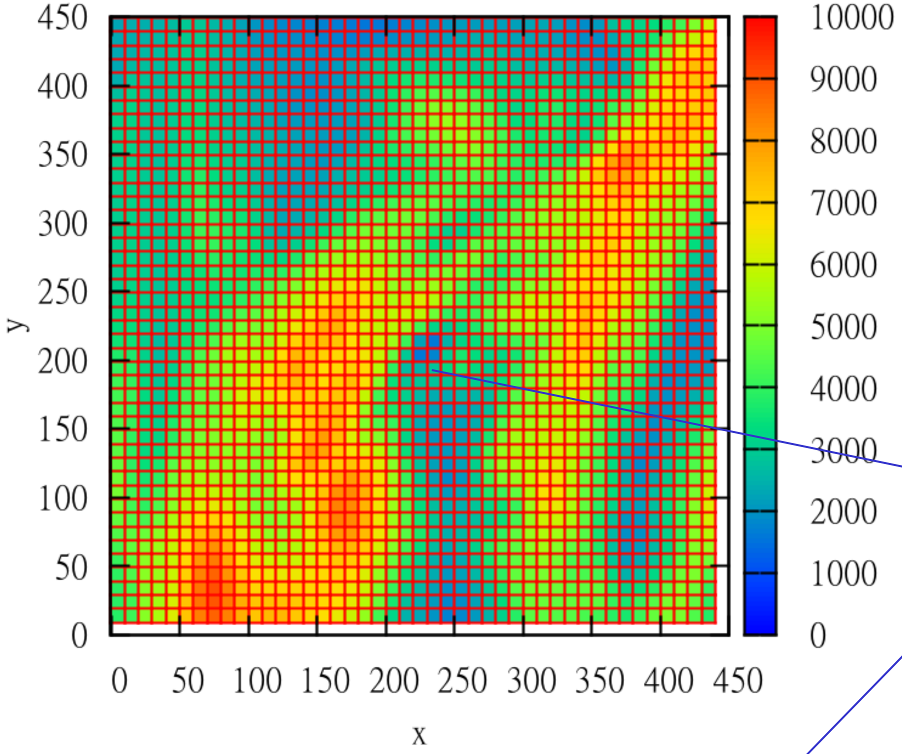


direct comparison of
pixel values

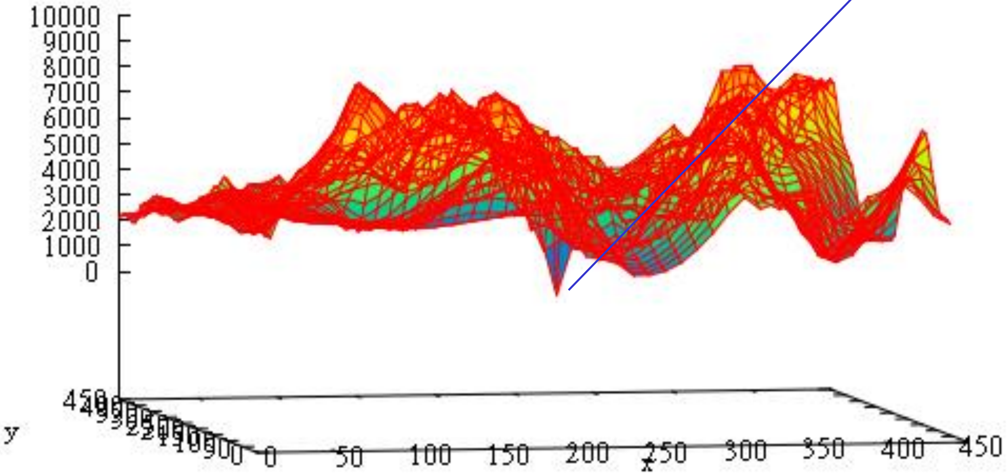


comparison of feature values
computed from images (e.g.
histograms, edge positions, …)

Direct Methods Illustrated



Minimum point of dissimilarity measure
(In this example, sum of squared difference of pixel intensities)



3)

Examples of Evaluation Functions

$$d_{\mathrm{SSD}}(x,y) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(T_{i,j} - I_{x+i,y+j})^2$$

: sum of squared differences (SSD)
→ min

$$d_{\mathrm{SAD}}(x,y) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}|T_{i,j} - I_{x+i,y+j}|$$

: sum of absolute differences (SAD)
→ min

$$C(x,y) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}T_{i,j}I_{x+i,y+j}$$

: cross correlation
→ max

average

$$C_{\mathrm{n}}(x,y) = \frac{\displaystyle\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(T_{i,j} - \bar{T})(I_{x+i,y+j} - \bar{I}_{x,y})}{\sqrt{\displaystyle\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(T_{i,j} - \bar{T})^2\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}(I_{x+i,y+j} - \bar{I}_{x,y})^2}}$$

: zero-mean normalized cross correlation (ZNCC)
→ max

Template Matching for Detection and Tracking

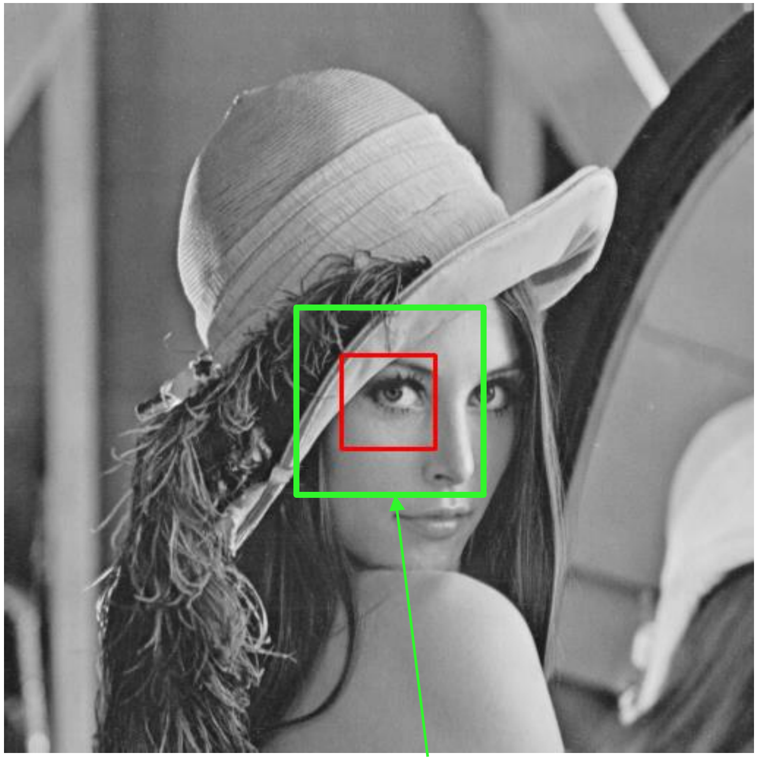
For Detection:

search area is set to the entire image

For Tracking:

search area is set at around the position in the previous frame (or a position predicted from previous frames)

input image



search area

template image

Implementation of SSD Matching

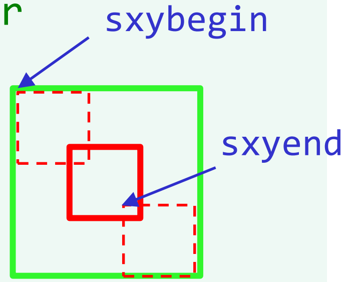
ic03_template_match_2d.py

```
def SSD(target, candidate):
    height, width = target.shape
    ssd_val = 0

    for j in range(height):
        for i in range(width):
            d = candidate[j, i] - target[j, i]
            ssd_val += d * d

    return ssd_val
```

```
min_ssd = sys.maxsize  ## initialized with a large, large number
for j in range(sybegin, syend):
    for i in range(sxbegin, sxend):
        candidate = image[j:(j + theight), i:(i + twidth)]
        ssd = SSD(target, candidate)
        if ssd < min_ssd:
            min_ssd = ssd
            min_location = (i, j)
```

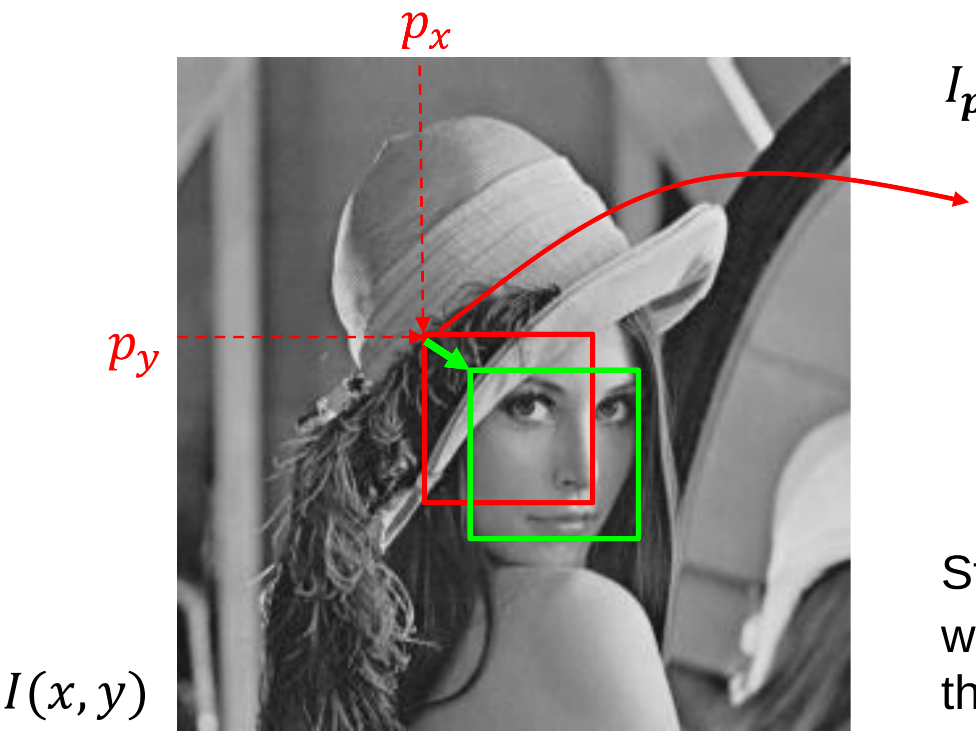


target shape: (theight, twidth)

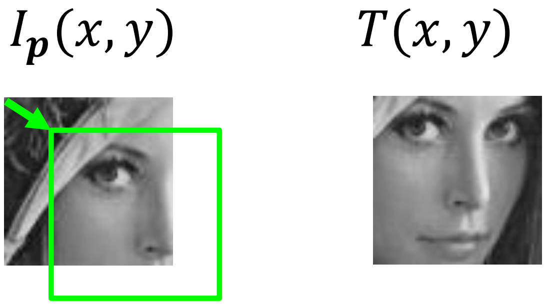
Gradient-based Optimization

Instead of brute force search for the minimum, let us consider application of Gauss-Newton optimization method to minimize:

$$\sum_{i,j} \{I(p_x + i, p_y + j) - T(i,j)\}^2$$



$$I_{\boldsymbol{p}}(x,y) = I(p_x + x, p_y + y)$$



Starting with an initial guess $\boldsymbol{p} = (p_x, p_y)$, we seek for $\Delta\boldsymbol{p} = (\Delta p_x, \Delta p_y)$ that makes $I_{\boldsymbol{p}+\Delta\mathbf{p}}(x,y)$ closer to $T(x,y)$

Lucas-Kanade Method: forward algorithm (1/2)

1st order Taylor expansion is applied:

$$\begin{aligned}E(\Delta p_x, \Delta p_y) &= \sum_{i,j} \{I_{\boldsymbol{p}}(\Delta p_x + i, \Delta p_y + j) - T(i,j)\}^2 \\ &\simeq \sum_{i,j} \left\{ I_{\boldsymbol{p}}(i,j) + \frac{\partial I_{\boldsymbol{p}}}{\partial x}(i,j)\Delta p_x + \frac{\partial I_{\boldsymbol{p}}}{\partial y}(i,j)\Delta p_y - T(i,j) \right\}^2 \\ &= \sum_{i,j} \left\{ \boxed{\frac{\partial I_{\boldsymbol{p}}}{\partial x}(i,j)}\Delta p_x + \boxed{\frac{\partial I_{\boldsymbol{p}}}{\partial y}(i,j)}\Delta p_y - \boxed{e_{\boldsymbol{p}}(i,j)} \right\}^2 \rightarrow \min_{\Delta p_x, \Delta p_y}\end{aligned}$$

$$e_{\boldsymbol{p}} = T - I_{\boldsymbol{p}}$$

$\dfrac{\partial I_{\boldsymbol{p}}}{\partial x}$

$\dfrac{\partial I_{\boldsymbol{p}}}{\partial y}$

$e_{\boldsymbol{p}}$

and partial derivatives are equated to 0:

$$\frac{\partial E}{\partial \Delta p_x} = 2\sum_{i,j} \left\{ \frac{\partial I_{\boldsymbol{p}}}{\partial x}(i,j)\Delta p_x + \frac{\partial I_{\boldsymbol{p}}}{\partial y}(i,j)\Delta p_y - e_{\boldsymbol{p}}(i,j) \right\} \frac{\partial I_{\boldsymbol{p}}}{\partial x}(i,j) = 0$$

$$\frac{\partial E}{\partial \Delta p_y} = 2\sum_{i,j} \left\{ \frac{\partial I_{\boldsymbol{p}}}{\partial x}(i,j)\Delta p_x + \frac{\partial I_{\boldsymbol{p}}}{\partial y}(i,j)\Delta p_y - e_{\boldsymbol{p}}(i,j) \right\} \frac{\partial I_{\boldsymbol{p}}}{\partial y}(i,j) = 0$$

Rearrainging them into linear equations with respect to $(\Delta p_x, \Delta p_y)$

$$\begin{pmatrix} \sum (\frac{\partial I_{\boldsymbol{p}}}{\partial x})^2 & \sum \frac{\partial I_{\boldsymbol{p}}}{\partial x}\frac{\partial I_{\boldsymbol{p}}}{\partial y} \\ \sum \frac{\partial I_{\boldsymbol{p}}}{\partial x}\frac{\partial I_{\boldsymbol{p}}}{\partial y} & \sum (\frac{\partial I_{\boldsymbol{p}}}{\partial y})^2 \end{pmatrix} \begin{pmatrix} \Delta p_x \\ \Delta p_y \end{pmatrix} = \begin{pmatrix} \sum \frac{\partial I_{\boldsymbol{p}}}{\partial x} e_{\boldsymbol{p}} \\ \sum \frac{\partial I_{\boldsymbol{p}}}{\partial y} e_{\boldsymbol{p}} \end{pmatrix}$$

Lucas-Kanade Method: forward algorithm (2/2)

- By solving the above equation, $(\Delta p_x, \Delta p_y)$ is only approximately best because of the 1st order Taylor approximation. We usually need to iteratively run the above process by updating

$$p_x \leftarrow p_x + \Delta p_x$$
$$p_y \leftarrow p_y + \Delta p_y$$

  and obtaining $I_{\boldsymbol{p}}(x, y) = I\left(p_x + x, p_y + y\right)$ with new $\boldsymbol{p} = (p_x, p_y)$

- Because $I_{\boldsymbol{p}}(x, y)$ changes, the derivatives and their products must be recomputed for each iteration

[Lucas and Kanade 1981]

Understanding in Vector Formulation

The problem to be solved is:

$$\|\boldsymbol{f}(\boldsymbol{p})-\boldsymbol{y}\|^2 \rightarrow \min_{\boldsymbol{p}}$$

Setting an initial guess of $\boldsymbol{p}$, we seek for additive update $\Delta\boldsymbol{p}$

$$E(\Delta\boldsymbol{p})=\left\|\boldsymbol{f}(\boldsymbol{p})+\frac{\partial \boldsymbol{f}(\boldsymbol{p})}{\partial \boldsymbol{p}}\Delta\boldsymbol{p}-\boldsymbol{y}\right\|^2=\|J\Delta\boldsymbol{p}-\boldsymbol{e}_{\boldsymbol{p}}\|^2 \rightarrow \min_{\Delta\boldsymbol{p}}$$

$$\frac{\partial E(\Delta\boldsymbol{p})}{\partial \Delta\boldsymbol{p}}=2J^T(J\Delta\boldsymbol{p}-\boldsymbol{e}_{\boldsymbol{p}})=\boldsymbol{0}^T$$

$$J^TJ\Delta\boldsymbol{p}=J^T\boldsymbol{e}_{\boldsymbol{p}}$$

After solving the above equation for $\Delta\boldsymbol{p}$, $\boldsymbol{p}$ is updated iteratively

$$\boldsymbol{p} \leftarrow \boldsymbol{p}+\Delta\boldsymbol{p}$$

$$\boldsymbol{f}_{\boldsymbol{p}}=\begin{pmatrix} I_{\boldsymbol{p}}(0,0)\\ I_{\boldsymbol{p}}(1,0)\\ I_{\boldsymbol{p}}(2,0)\\ \vdots \\ I_{\boldsymbol{p}}(0,n-1)\end{pmatrix}$$

$$\boldsymbol{y}=\begin{pmatrix} T(0,0)\\ T(1,0)\\ T(2,0)\\ \vdots \\ T(0,n-1)\end{pmatrix}$$

J^TJ

(Gauss-Newton approximation of) Hessian matrix

$$J=\begin{pmatrix} \frac{\partial I_{\boldsymbol{p}}(0,0)}{\partial x} & \frac{\partial I_{\boldsymbol{p}}(0,0)}{\partial y}\\ \frac{\partial I_{\boldsymbol{p}}(1,0)}{\partial x} & \frac{\partial I_{\boldsymbol{p}}(1,0)}{\partial y}\\ \vdots & \vdots \end{pmatrix}$$

Jacobian matrix

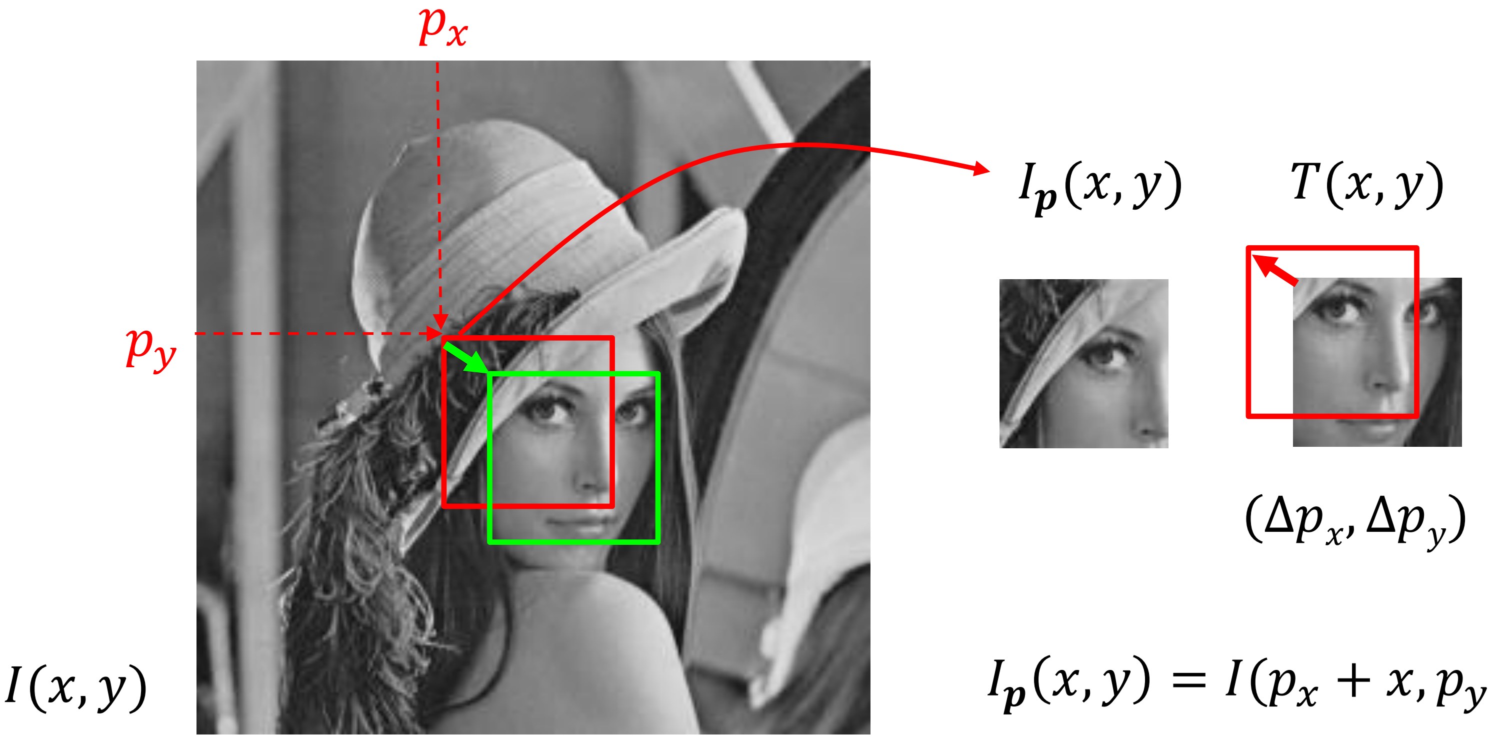
$$\boldsymbol{e}_{\boldsymbol{p}}=\begin{pmatrix} T(0,0)-I_{\boldsymbol{p}}(0,0)\\ T(1,0)-I_{\boldsymbol{p}}(1,0)\\ \vdots \end{pmatrix}$$

error vector

Inverse Algorithm (1/2)

The recomputation of derivatives and their products per iteration can be avoided by exchanging the role of T and $I_{\boldsymbol{p}}$

$$E(\Delta p_x, \Delta p_y) = \sum_{i,j} \left\{ T(\Delta p_x + i, \Delta p_y + j) - I_{\boldsymbol{p}}(i,j) \right\}^2$$



$$I_{\boldsymbol{p}}(x,y) = I(p_x + x, p_y + y)$$

Inverse Algorithm (2/2)

$$E(\Delta p_x, \Delta p_y) \simeq \sum_{i,j} \left\{ T(i,j) + \frac{\partial T}{\partial x}(i,j)\Delta p_x + \frac{\partial T}{\partial y}(i,j)\Delta p_y - I_{\boldsymbol{p}}(i,j) \right\}^2$$

$$e_{\boldsymbol{p}} = I_{\boldsymbol{p}} - T$$

$$= \sum_{i,j} \left\{ \frac{\partial T}{\partial x}(i,j)\Delta p_x + \frac{\partial T}{\partial y}(i,j)\Delta p_y - e_{\boldsymbol{p}}(i,j) \right\}^2 \to \min_{\Delta p_x, \Delta p_y}$$

$$\begin{pmatrix} \sum \left(\frac{\partial T}{\partial x}\right)^2 & \sum \frac{\partial T}{\partial x}\frac{\partial T}{\partial y} \\ \sum \frac{\partial T}{\partial x}\frac{\partial T}{\partial y} & \sum \left(\frac{\partial T}{\partial y}\right)^2 \end{pmatrix} \begin{pmatrix} \Delta p_x \\ \Delta p_y \end{pmatrix} = \begin{pmatrix} \sum \frac{\partial T}{\partial x} e_{\boldsymbol{p}} \\ \sum \frac{\partial T}{\partial y} e_{\boldsymbol{p}} \end{pmatrix}$$

After solving $(\Delta p_x, \Delta p_y)$, we resample $I_{\boldsymbol{p}}(x, y)$ with new $\boldsymbol{p}$ updated by

$$p_x \leftarrow p_x - \Delta p_x$$
$$p_y \leftarrow p_y - \Delta p_y$$

Move I_p in the *opposite* direction

Implementation of the Inverse LK (1/2)

ic03_lucas_kanade_2d.py

```
for j in range(1, theight - 1):
    for i in range(1, twidth - 1):
        Tx[j, i] = (T[j, i + 1] - T[j, i - 1]) / 2
        Ty[j, i] = (T[j + 1, i] - T[j - 1, i]) / 2
        TxTx[j, i] = Tx[j, i] * Tx[j, i]
        TyTy[j, i] = Ty[j, i] * Ty[j, i]
        TxTy[j, i] = Tx[j, i] * Ty[j, i]
        H[0, 0] += TxTx[j, i]
        H[1, 1] += TyTy[j, i]
        H[0, 1] += TxTy[j, i]
H[1, 0] = H[0, 1]
```

```
for j in range(1, theight - 1):
    for i in range(1, twidth - 1):
        err[j, i] = Ip[j, i] - T[j, i]
        Tx_err[j, i] = Tx[j, i] * err[j, i]
        Ty_err[j, i] = Ty[j, i] * err[j, i]
        Jt_err[0] += Tx_err[j, i]
        Jt_err[1] += Ty_err[j, i]
```

Implementation of the Inverse LK (2/2)

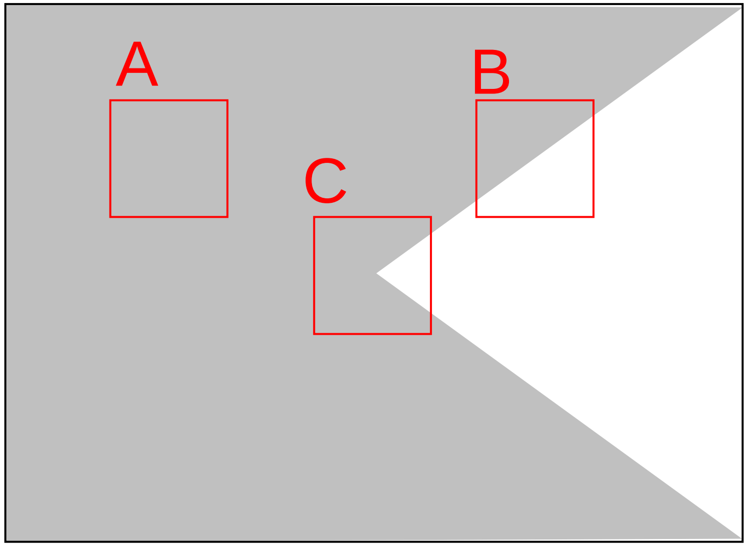
```
def match_template_lk(image, current_center, T, Tx, Ty, JtJ, max_iter=50):
    theight, twidth = T.shape

    for iter in range(max_iter):
        Ip = cv2.getRectSubPix(image, (twidth, theight), current_center)
        Ip = np.float32(Ip)
        Jt_err = compute_Jt_err(Ip, T, Tx, Ty)
        dp = np.linalg.solve(JtJ, Jt_err)
        current_center = (current_center[0] - dp[0], current_center[1] - dp[1])
        if np.linalg.norm(dp) < 0.2:
            break

    return current_center
```

Feature Point Detection

Let’s consider a case where we need to automatically extract some (often many) points to be tracked to analyze e.g. the scene structure or motion



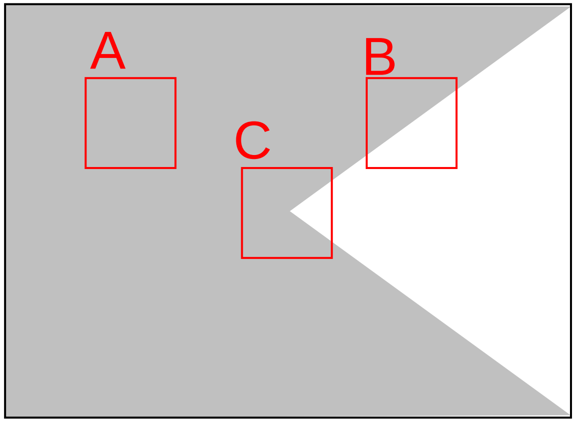
A: Block with constant intensity is not suitable
B: Block including only edges with the same direction is also not suitable
C: Suitable for tracking

How to find blocks like C?

Analysis of Hessian Matrix

$$\begin{pmatrix} \sum(\frac{\partial T}{\partial x})^2 & \sum \frac{\partial T}{\partial x}\frac{\partial T}{\partial y} \\ \sum \frac{\partial T}{\partial x}\frac{\partial T}{\partial y} & \sum(\frac{\partial T}{\partial y})^2 \end{pmatrix} \begin{pmatrix} \Delta p_x \\ \Delta p_y \end{pmatrix} = \begin{pmatrix} \sum \frac{\partial T}{\partial x} e_{\boldsymbol{p}} \\ \sum \frac{\partial T}{\partial y} e_{\boldsymbol{p}} \end{pmatrix}$$

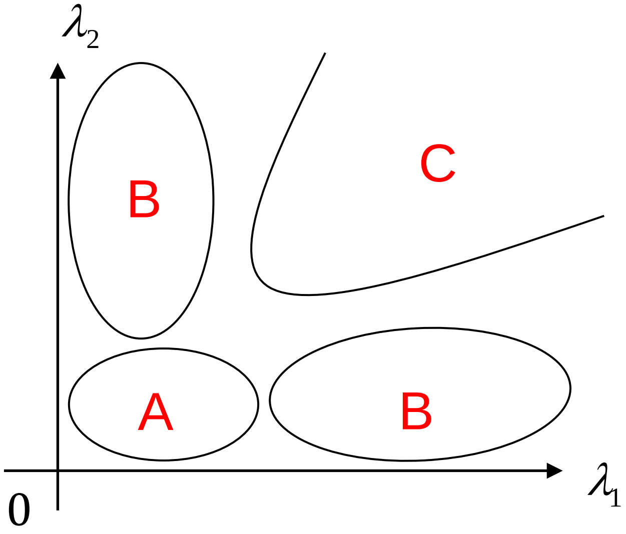
The above equation should be stably solved for a block suitable for tracking



By Diagonalizing $J^T J = Q^{-1} \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} Q$, we have

$$\begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} Q\Delta\boldsymbol{p} = QJ^T \boldsymbol{e}_{\boldsymbol{p}}$$

(Since $J^T J$ is positive semi-definite symmetric matrix, $\lambda_1, \lambda_2 \geq 0$ and Q is orthogonal matrix)



- Both λ_1 and λ_2 should be sufficiently larger than zero
- Too small λ_i implies that determining i-th element of $Q\Delta\boldsymbol{p}$ is difficult

Examples of Feature Point Detector

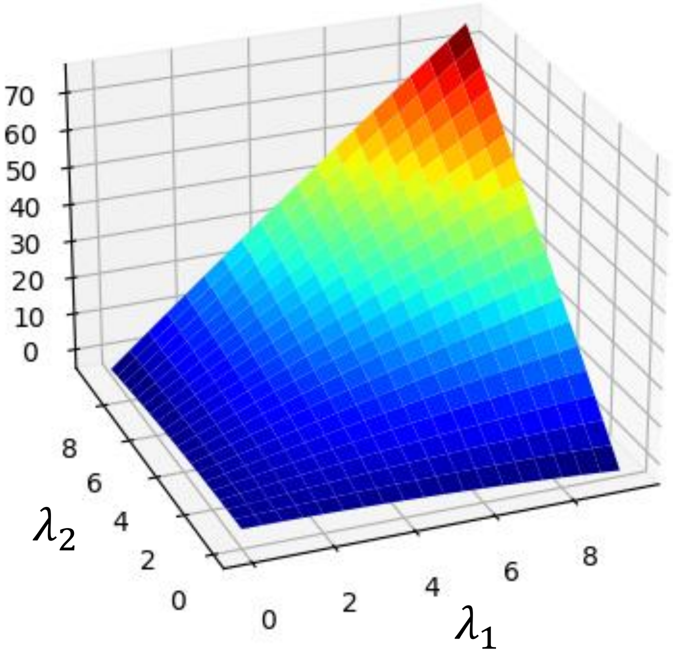
Good Features to Track [Tomasi and Kanade 1991]

$$\min(\lambda_1, \lambda_2)$$

Harris operator [Harris and Stephens 1988]

$$\det H - k(\operatorname{tr} H)^2$$
$$= \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2$$

$\lambda_1 \lambda_2 - 0.04(\lambda_1 + \lambda_2)^2$



The points with large values of the above indicators, which are “good” for tracking and/or matching, are called feature point, interest point, corner point, keypoint and so on.

Implementation of Feature Point Detectors (1/2)

ic03_feature_points.py

```
def hessian_map(T, block_size=5):
    Tx = np.gradient(T, axis=1)
    Ty = np.gradient(T, axis=0)
    TxTx = Tx * Tx
    TyTy = Ty * Ty
    TxTy = Tx * Ty

    theight = T.shape[0]
    twidth = T.shape[1]
    H = np.zeros((theight, twidth, 2, 2), dtype=T.dtype)
    H[:, :, 0, 0] = cv2.blur(TxTx, (block_size, block_size))
    H[:, :, 1, 1] = cv2.blur(TyTy, (block_size, block_size))
    H[:, :, 0, 1] = cv2.blur(TxTy, (block_size, block_size))
    H[:, :, 1, 0] = H[:, :, 0, 1]

    return H
```

Gradients are computed for all over the image (to avoid recomputing them for the same point again and again)

Tensor of order 4; if you are not familiar with tensors, imagine a `theight` × `twidth` array whose element is 2×2 matrix

Implementation of Feature Point Detectors (2/2)

```
def min_eigen_value_map(H):
    a = H[:, :, 0, 0]  # H = [a b]
    b = H[:, :, 0, 1]  #     [c d]
    c = H[:, :, 1, 0]
    d = H[:, :, 1, 1]

    ## the smaller solution of s^2 - (a + d) s + ad - bc = 0
    min_eig = ((a + d) - np.sqrt((a - d)**2 + 4 * b * c)) / 2

    return min_eig
```

```
def harris_map(H, coeff_k):
    a = H[:, :, 0, 0]  # H = [a b]
    b = H[:, :, 0, 1]  #     [c d]
    c = H[:, :, 1, 0]
    d = H[:, :, 1, 1]

    return (a * d - b * c) - coeff_k * (a + d)**2
```

Other Feature Point Detectors

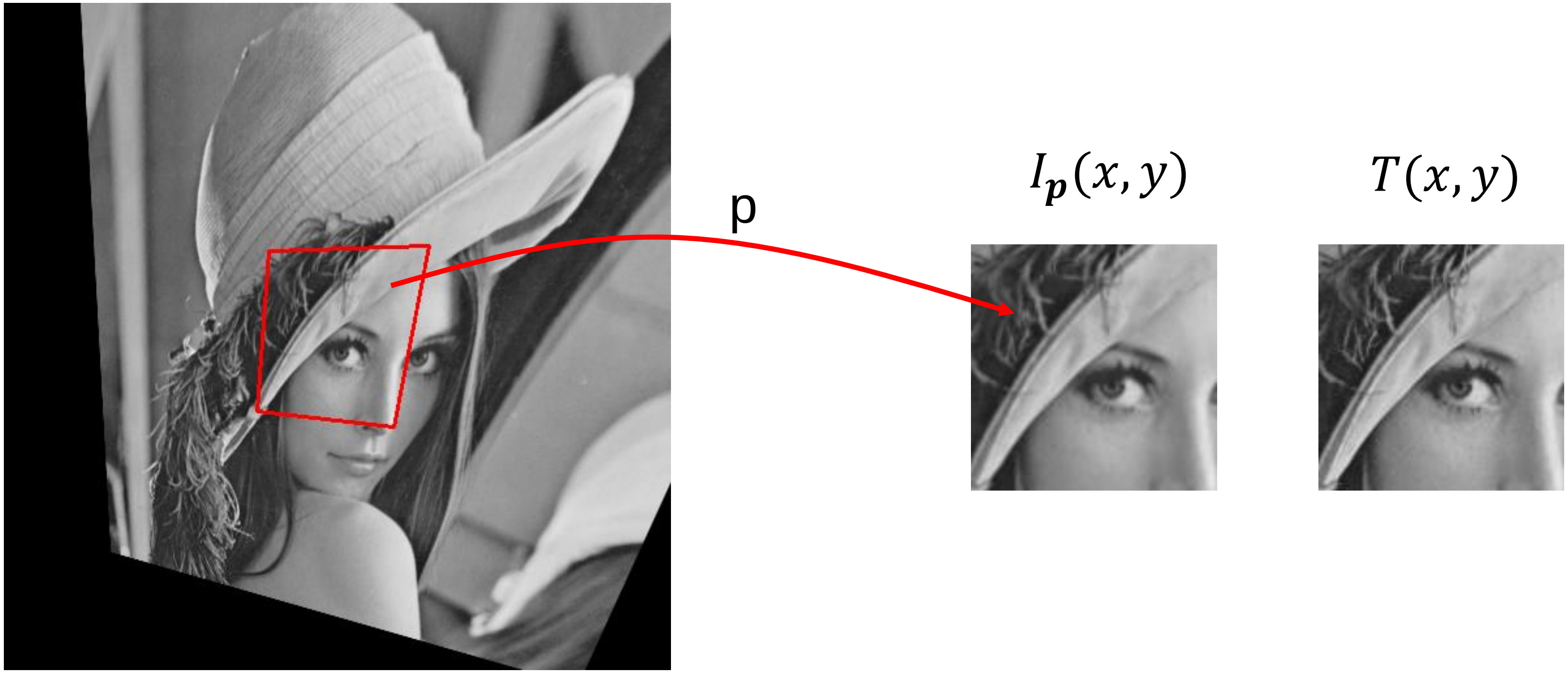
SIFT detector [Lowe 2004]

- Build a Gaussian scale space and apply (an approximate) Laplacian operator in each scale
- Detect extrema of the results (i.e. strongest responses among their neighbor in space as well as in scale)
- Eliminate edge responses
- (Often followed by encoding of edge orientation histogram in the neighborhood into a fixed-size vector, called a feature point descriptor, which can be compared with each other by Euclidean distance)

FAST detector [Rosten et al. 2010]

- Heuristics based on pixel values along a surrounding circle
- Optimized for speed and quality by machine learning approach

Generalization to Different Warps

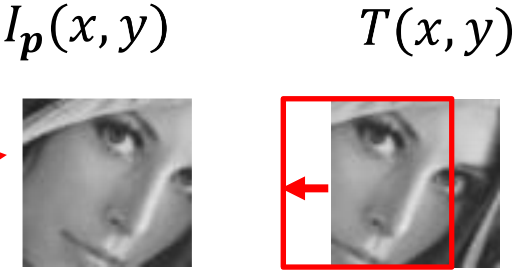


We want to generalize the inverse algorithm of Lucas-Kanade method for warps beyond 2D translation

Naïve (and wrong) Generalization

Let's think of the rigid transform case where $\boldsymbol{p} = (p_x, p_y, p_\theta)$

$$E(\Delta \boldsymbol{p}) \simeq \sum_{i,j} \left\{ \frac{\partial T}{\partial p_x}(i,j)\Delta p_x + \frac{\partial T}{\partial p_y}(i,j)\Delta p_y + \frac{\partial T}{\partial p_\theta}(i,j)\Delta p_\theta - e_{\boldsymbol{p}}(i,j) \right\}^2 \quad \to \min_{\Delta \boldsymbol{p}}$$



$\Delta(p_x, p_y, p_\theta) = (-10, 0, 0)$

Then, should we update $\boldsymbol{p}$ as
$\boldsymbol{p} \leftarrow \boldsymbol{p} - \Delta \boldsymbol{p}$?
Obviously no!

What was wrong?

What we must do is to invert the warp, which *happened* to be equal to negating the signs of parameters in the translation case:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \quad \underset{\substack{\text{correct} \\ \text{inverse}}}{\Longleftrightarrow} \quad \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

However, it generally does not

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos p_\theta & -\sin p_\theta & p_x \\ \sin p_\theta & \cos p_\theta & p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \neq \begin{pmatrix} \cos(-p_\theta) & -\sin(-p_\theta) & -p_x \\ \sin(-p_\theta) & \cos(-p_\theta) & -p_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

So, what to do?

First, we need to introduce the warping function explicitly:

$$\boldsymbol{x}' = \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x}) \qquad \boldsymbol{x} = (x, y)^T,\ \boldsymbol{x}' = (x', y')^T$$

$$E(\Delta \boldsymbol{p}) = \sum_{\boldsymbol{x}} \left\{ T(\boldsymbol{w}_{\Delta \boldsymbol{p}}(\boldsymbol{x})) - I(\boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x})) \right\}^2 \to \min_{\Delta \boldsymbol{p}}$$

cf. the translation case: $E(\Delta \boldsymbol{p}) = \sum_{i,j} \left\{ T(\Delta p_x + i, \Delta p_y + j) - I_{\boldsymbol{p}}(i,j) \right\}^2$

$$E(\Delta \boldsymbol{p}) \simeq \sum_{i,j} \left\{ \left(\frac{\partial T}{\partial p_1}(i,j)\Delta p_1 + \frac{\partial T}{\partial p_2}(i,j)\Delta p_2 + \cdots \right) - e_{\boldsymbol{p}}(i,j) \right\}^2$$

$$\frac{\partial T}{\partial p_k}(x,y) = \left. \frac{\partial}{\partial p_k} T(\boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x})) \right|_{\boldsymbol{p}=\boldsymbol{0}}$$

$$= \left. \frac{\partial T}{\partial \boldsymbol{x}} \frac{\partial \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x})}{\partial p_k} \right|_{\boldsymbol{p}=\boldsymbol{0}}$$

How much the pixel value changes when the pixel coordinates move

How much the pixel coordinates move when p_k moves around 0

Warp Functions and Their Derivatives

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & t_x \\ \sin\theta & \cos\theta & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

rigid transform: $\boldsymbol{p} = (t_x, t_y, \theta)$

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x}) = \begin{pmatrix} x\cos\theta - y\sin\theta + t_x \\ x\sin\theta + y\cos\theta + t_y \end{pmatrix}$$

$$\left.\frac{\partial}{\partial \boldsymbol{p}} \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x})\right|_{\boldsymbol{p}=\boldsymbol{0}} = \begin{pmatrix} 1 & 0 & -y \\ 0 & 1 & x \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \propto \begin{pmatrix} 1+p_1 & p_2 & p_3 \\ p_4 & 1+p_5 & p_6 \\ p_7 & p_8 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

homography transform

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x}) = \begin{pmatrix} \dfrac{(1+p_1)x + p_2 y + p_3}{p_7 x + p_8 y + 1} \\ \dfrac{p_4 x + (1+p_5)y + p_6}{p_7 x + p_8 y + 1} \end{pmatrix}$$

$$\left.\frac{\partial}{\partial \boldsymbol{p}} \boldsymbol{w}_{\boldsymbol{p}}(\boldsymbol{x})\right|_{\boldsymbol{p}=\boldsymbol{0}} = \begin{pmatrix} x & y & 1 & 0 & 0 & 0 & -x^2 & -xy \\ 0 & 0 & 0 & x & y & 1 & -xy & -y^2 \end{pmatrix}$$

Inverse Compositional Algorithm of LK Method

Precompute J and $J^T J$ once template is given

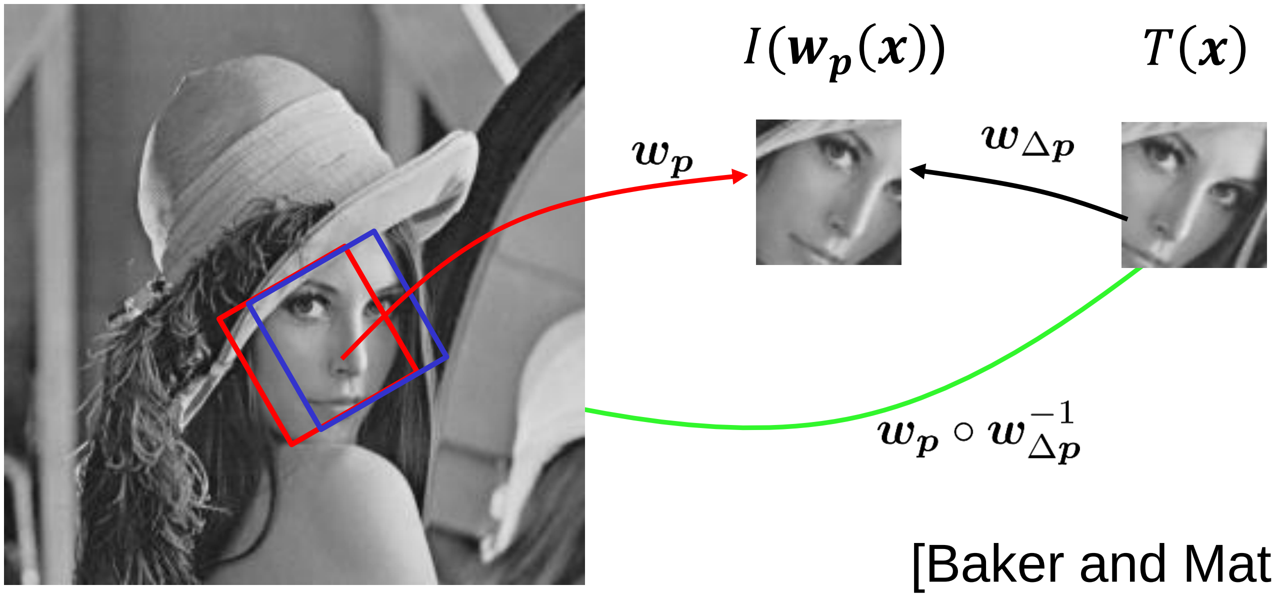
Iteratively solve $J^T J\,\Delta\boldsymbol{p} = J^T \boldsymbol{e}_{\boldsymbol{p}}$ and update the warp by composing the obtained incremental warp $\boldsymbol{w}_{\Delta\boldsymbol{p}}$

$$\boldsymbol{w}_{\boldsymbol{p}} \leftarrow \boldsymbol{w}_{\boldsymbol{p}} \circ \boldsymbol{w}_{\Delta\boldsymbol{p}}^{-1}$$

$$J = \begin{pmatrix} \frac{\partial T(0,0)}{\partial p_1} & \frac{\partial T(0,0)}{\partial p_2} & \cdots \\ \frac{\partial T(1,0)}{\partial p_1} & \frac{\partial T(1,0)}{\partial p_2} & \cdots \\ \vdots & \vdots & \end{pmatrix}$$

pixels

parameters



[Baker and Matthews 2004]

Implementation for Homography Warp (1/2)

ic03_lucas_kanade_homography.py

```
def compute_derivatives(T):
    theight = T.shape[0]
    twidth = T.shape[1]
    npix = twidth * theight
    Tx = np.gradient(T, axis=1).reshape(npix, 1)
    Ty = np.gradient(T, axis=0).reshape(npix, 1)

    dwdp_x = np.empty((npix, 8), dtype=T.dtype)
    dwdp_y = np.empty((npix, 8), dtype=T.dtype)
    row = 0
    for y in range(theight):
        for x in range(twidth):
            dwdp_x[row] = np.array([ x, y, 1, 0, 0, 0, -x*x, -x*y ])
            dwdp_y[row] = np.array([ 0, 0, 0, x, y, 1, -x*y, -y*y ])
            row += 1

    J = Tx * dwdp_x + Ty * dwdp_y
    JtJ = np.dot(J.T, J)
    return J, JtJ
```

row-wise multiply and element-wise add

Implementation for Homography Warp (2/2)

current guess is passed as a homography matrix

```
def track_homography_lk(image, homography_p, T, J, JtJ, max_iter=50):
    theight, twidth = T.shape
    npix = twidth * theight

    for iter in range(max_iter):
        Ip = cv2.warpPerspective(image, inv(homography_p), (twidth, theight))
        Ip = np.float32(Ip)
        err = (Ip - T).reshape(npix)
        dp = np.linalg.solve(JtJ, np.dot(J.T, err))
        homography_dp = np.array([[1 + dp[0], dp[1], dp[2]],
                                  [dp[3], 1 + dp[4], dp[5]],
                                  [dp[6], dp[7], 1.0]])
        homography_p = np.dot(homography_p, inv(homography_dp))

    return homography_p
```

composition of warps is done by matrix multiplication

returns an updated homography matrix

Other Choices of Optimization Methods

Levenberg-Marquardt method

$$(J^T J + \mu I)\Delta \boldsymbol{p} = J^T \boldsymbol{e}_{\boldsymbol{p}}$$

I : identity matrix
μ : scalar coefficient (updated between iterations)
(small μ : more like Gauss-Newton,
large μ : more like steepest descent)

Efficient Second-order Minimization method [Banhimane and Malis 2007]

$$(J^T J)\Delta \boldsymbol{p} = J^T \boldsymbol{e}_{\boldsymbol{p}}, \ J = (J_1 + J_2)/2$$

J_1: derivative of template image
J_2: derivative of current warped image
(Possible when parametrized with special care)

Exercises (Not Assignments)

Copy and modify ic03_lucas_kanade_homography.py to apply a simpler version of Levenberg-Marquardt method in which μ is fixed, i.e., replace `JtJ` for example with `JtJ + 0.001 * np.eye(8)` in:

```
dp = np.linalg.solve(JtJ, np.dot(J.T, err))
```

You may want to choose different μ other than 0.001 and see the difference. You may also need to increase `max_iter`.

Copy and modify ic03_lucas_kanade_homography.py to visualize `J` (Jacobian matrix).

Hint:

- `J[:, k]` (k-th column of J) gives derivative with respect to the k-th parameter, which should be reshaped to the shape of the template image
- The values should be normalized to fit [0, 1] when passed to `cv2.imshow`

References

- B. K. P. Horn and B. G. Schunck: Determining Optical Flow, Artificial Intelligence, vol.17, pp.185-203, 1981.
- C. Harris and M. Stephens: A Combined Corner and Edge Detector, Proc. 14th Alvey Vision Conference, pp.147-151, 1988.
- B. D. Lucas and T. Kanade: An Iterative Image Registration Technique with an Application to Stereo Vision, Proc. 7th International Conference on Artificial Intelligence, pp.674-679, 1981.
- S. Baker and I. Matthews: Lucas-Kanade 20 Years On: A Unifying Framework, International Journal of Computer Vision, vol. 56, no. 3, 2004.
- S. Benhimane and E. Malis: Homography-based 2D Visual Tracking and Servoing, International Journal of Robotics Research, vol. 26, no. 7, pp.661-676, 2007.
- D. Lowe, Distinctive Image Features from Scale-Invariant Keypoints, International Journal of Computer Vision, vol. 60, no. 2, pp. 91-110, 2004.
- E. Rosten, R. Porter and T. Drummond: Faster and Better: A Machine Learning Approach to Corner Detection, IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 32, no. 1, pp. 105-119, 2010.
- C. Tomasi and T. Kanade: Detection and Tracking of Point Features, Shape and Motion from Image Streams: a Factorization Method –Part 3, Technical Report CMU-CS-91-132, Carnegie Mellon University, 1991.