HOMEWORK 6: OBJECT TRACKING IN VIDEOS

16-720B Introduction to Computer Vision (Fall 2020)
Carnegie Mellon University
Homework PDF Link
BY: Feng Xiang[*]
DUE: Monday, December 7, 2020 11:59 PM

Notes

- I worked on this project with Tom Xu and Gerald D'Ascoli

[*]Compiled on Monday 7th December, 2020 at 02:43

Question 03

Q1.1

Background An affine warp matrix is defined as the following matrix transformation:

$$W(p) = \begin{bmatrix} 1+p_1 & p_3 & p_5 \\ p_2 & 1+p_4 & p_6 \\ 0 & 0 & 1 \end{bmatrix}$$

Given a set of x and y coordinates of a pixel on an image, the warp transformed coordinates of the original set can be viewed as the following warped function:

$$W(x;p) = \begin{bmatrix} p_1x + p_3y + p_5 \\ p_2x + p_4y + p_6 \end{bmatrix}$$

Answer The Jacobian Matrix in this image alignment algorithm is defined as the matrix of partial derivatives of the warp function equation with respect to each warp parameter. The matrix shown below describes the content of each element of the Jacobian Matrix:

$$\frac{\partial W}{\partial p} = \begin{bmatrix} \frac{\partial W_x}{\partial p_1} & \frac{\partial W_x}{\partial p_2} & \frac{\partial W_x}{\partial p_3} & \frac{\partial W_x}{\partial p_4} & \frac{\partial W_x}{\partial p_5} & \frac{\partial W_x}{\partial p_6} \\ \frac{\partial W_y}{\partial p_1} & \frac{\partial W_y}{\partial p_2} & \frac{\partial W_y}{\partial p_3} & \frac{\partial W_y}{\partial p_4} & \frac{\partial W_y}{\partial p_5} & \frac{\partial W_y}{\partial p_6} \end{bmatrix}$$

Given the matrix equation for the warp transform as described earlier, the partial derivatives for the x-coordinates of the warp transformation are simplified to the following:

$$\frac{\partial W_x}{\partial p_1} = x$$
$$\frac{\partial W_x}{\partial p_2} = 0$$
$$\frac{\partial W_x}{\partial p_3} = y$$
$$\frac{\partial W_x}{\partial p_4} = 0$$
$$\frac{\partial W_x}{\partial p_5} = 1$$
$$\frac{\partial W_x}{\partial p_6} = 0$$

Likewise, the partial derivatives for the y-coordinates of the warp transformation are simplified to the following:

$$\frac{\partial W_y}{\partial p_1} = 0$$
$$\frac{\partial W_y}{\partial p_2} = x$$
$$\frac{\partial W_y}{\partial p_3} = 0$$
$$\frac{\partial W_y}{\partial p_4} = y$$
$$\frac{\partial W_y}{\partial p_5} = 0$$
$$\frac{\partial W_y}{\partial p_6} = 1$$

The resulting Jacobian Matrix is defined to be:

$$\frac{\partial W}{\partial p} = \begin{bmatrix} x & 0 & y & 0 & 1 & 0 \\ 0 & x & 0 & y & 0 & 1 \end{bmatrix}$$

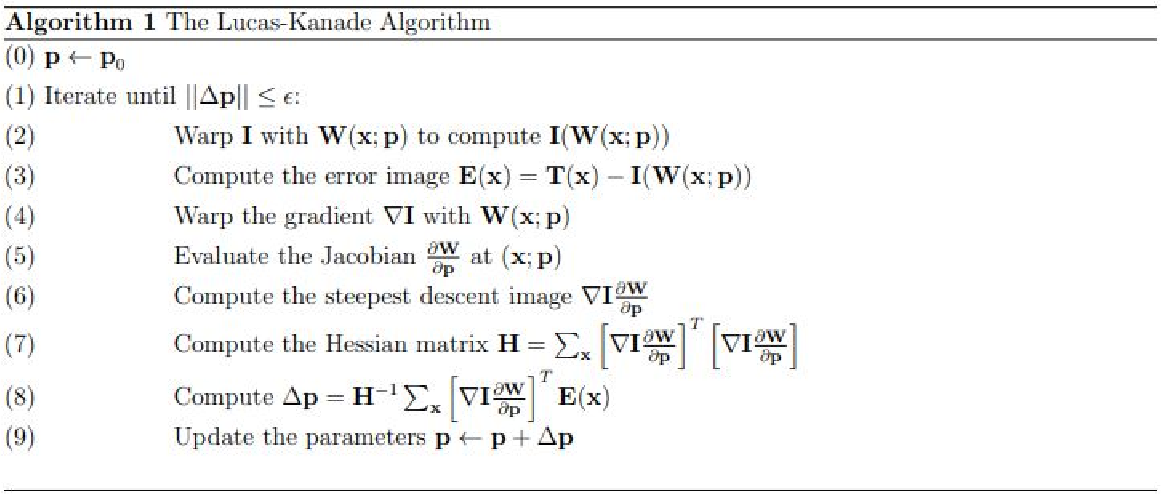
References

- Lecture Slides: Image Alignment - Lucas Kanade

Q1.2

Background N/A

Answer The figure shown below (see Figure 0.1) displays the general steps taken in the Lucas-Kanade image tracking algorithm:



```
Algorithm 1 The Lucas-Kanade Algorithm
(0) $\mathbf{p} \leftarrow \mathbf{p}_0$
(1) Iterate until $||\Delta\mathbf{p}|| \leq \epsilon$:
(2)         Warp $\mathbf{I}$ with $\mathbf{W}(\mathbf{x};\mathbf{p})$ to compute $\mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p}))$
(3)         Compute the error image $\mathbf{E}(\mathbf{x}) = \mathbf{T}(\mathbf{x}) - \mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p}))$
(4)         Warp the gradient $\nabla\mathbf{I}$ with $\mathbf{W}(\mathbf{x};\mathbf{p})$
(5)         Evaluate the Jacobian $\frac{\partial\mathbf{W}}{\partial\mathbf{p}}$ at $(\mathbf{x};\mathbf{p})$
(6)         Compute the steepest descent image $\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}$
(7)         Compute the Hessian matrix $\mathbf{H} = \sum_{\mathbf{x}} \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right]^T \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right]$
(8)         Compute $\Delta\mathbf{p} = \mathbf{H}^{-1}\sum_{\mathbf{x}} \left[\nabla\mathbf{I}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right]^T \mathbf{E}(\mathbf{x})$
(9)         Update the parameters $\mathbf{p} \leftarrow \mathbf{p} + \Delta\mathbf{p}$
```

Figure 0.1: Given Layout of Lucas-Kanade Algorithm

To analyze the big *O* notation of each step taken, one can consider the variables being considered in the algorithm step and how that computation complexity will change if *n*,*m*, and *p* were increased or decreased (where *n* is the number of pixels in the template, *m* is the number of pixels in the image, and *p* is the number of warp parameters chosen).

The table shown below displays the determined big *O* notation at each algorithm step:

Question 1.2 Determined Big O Notation	
Step i	$O(\_\_\_)$
(2)	$O(mp)$
(3)	$O(m)$
(4)	$O(mp)$
(5)	$O(mp)$
(6)	$O(mp)$
(7)	$O(p^2m)$
(8)	$O(p^3)$
(9)	$O(p)$

The from the determined big *O* notations for each algorithm line in the iteration loop, it is determined that the big *O* notation of the Lucas-Kanade alogrithm is on the scale of the third power, or more specifically:

$$O(p^2m + p^3)$$

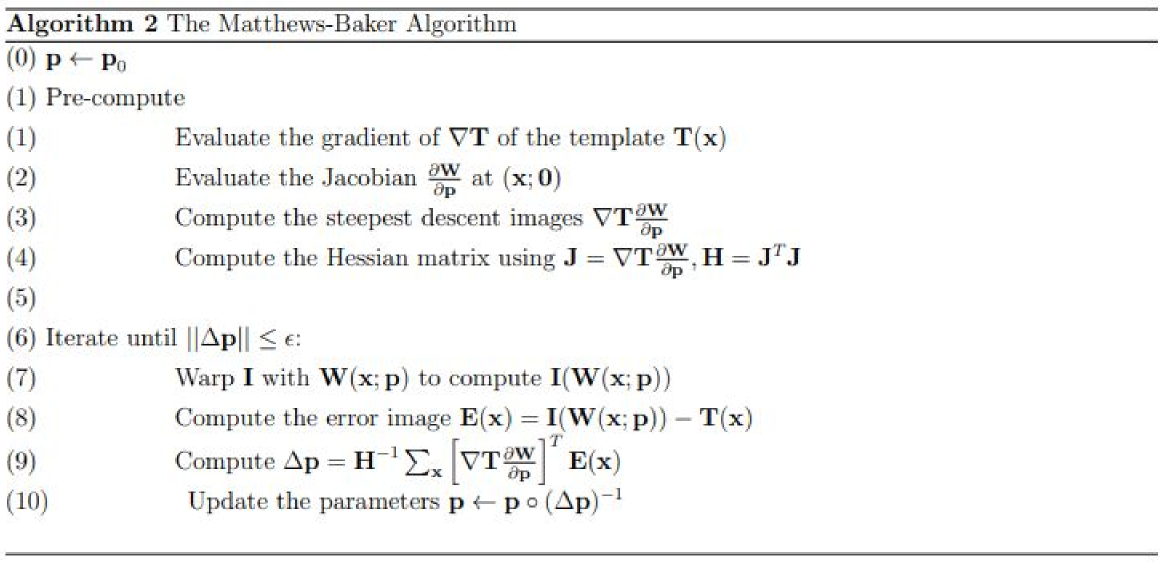
References

- Scholarly Article: Lucas-Kanade 20 Years On

Q1.3

Background N/A

Answer The figure shown below (see Figure 0.2) displays the general steps taken in the Matthews-Baker image tracking algorithm:



```
Algorithm 2 The Matthews-Baker Algorithm
(0) $\mathbf{p} \leftarrow \mathbf{p}_0$
(1) Pre-compute
(1)         Evaluate the gradient of $\nabla\mathbf{T}$ of the template $\mathbf{T}(\mathbf{x})$
(2)         Evaluate the Jacobian $\frac{\partial\mathbf{W}}{\partial\mathbf{p}}$ at $(\mathbf{x};\mathbf{0})$
(3)         Compute the steepest descent images $\nabla\mathbf{T}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}$
(4)         Compute the Hessian matrix using $\mathbf{J} = \nabla\mathbf{T}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}, \mathbf{H} = \mathbf{J}^T\mathbf{J}$
(5)
(6) Iterate until $||\Delta\mathbf{p}|| \leq \epsilon$:
(7)         Warp $\mathbf{I}$ with $\mathbf{W}(\mathbf{x};\mathbf{p})$ to compute $\mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p}))$
(8)         Compute the error image $\mathbf{E}(\mathbf{x}) = \mathbf{I}(\mathbf{W}(\mathbf{x};\mathbf{p})) - \mathbf{T}(\mathbf{x})$
(9)         Compute $\Delta\mathbf{p} = \mathbf{H}^{-1}\sum_{\mathbf{x}}\left[\nabla\mathbf{T}\frac{\partial\mathbf{W}}{\partial\mathbf{p}}\right]^T\mathbf{E}(\mathbf{x})$
(10)         Update the parameters $\mathbf{p} \leftarrow \mathbf{p} \circ (\Delta\mathbf{p})^{-1}$
```

Figure 0.2: Given Layout of Matthews-Baker Algorithm

To analyze the big O notation of each step taken, one can consider the varables being considered in the alogoritm step and how that computation complexity will change if *n*,*m*, and *p* were increased or decreased (where *n* is the number of pixels in the template, *m* is the number of pixels in the image, and *p* is the number of warp parameters chosen).

The table shown below displays the determined big O notation at each algorithm step:

Question 1.2 Determined Big O Notation	
Step i	$O(\_\_\_)$
(7)	$O(mp)$
(8)	$O(m)$
(9)	$O(p^3)$
(10)	$O(p^2)$

Observing the determined big O notations at each algorithm step in the iteration loop, one can determined that though the result big O notation is still on the scale of the third power, the big O notation pertaining to the pixel size of the template or image has been reduced. One can assume that the computation complexity with the total number of pixels of the template or image is greater that the number of warp parameters used.

The resultant big O notation was determined to be:

$$O(pm + p^3)$$

References

- Scholarly Article: Lucas-Kanade 20 Years On

Question 04

Q2.1

Background N/A

Answer The code implementation for the question is shown below:

```
import numpy as np
from scipy.interpolate import RectBivariateSpline

def LucasKanade(It, It1, rect):
    # Input:
    #   It: template image
    #   It1: Current image
    #   rect: Current position of the object
    #   (top left, bot right coordinates: x1, y1, x2, y2)
    # Output:
    #   p: movement vector dx, dy

    # set up the threshold
    threshold = 0.01875
    maxIters = 100
    p = np.zeros(2)
    x1,y1,x2,y2 = rect

    # put your implementation here

    #define magnitude of delta p to be well above the threshold to start
    mag_del_p = threshold + 10
    #define iteration number to be zero to start
    i = 0

    #create windows segment from template image based on rect box
    It_rect = It[rect[1]:rect[3]+1,rect[0]:rect[2]+1]

    onesI_mat = np.ones_like(It_rect)
    Iindices = np.array(np.where(onesI_mat>=0))
    #offset rows by y value of rect
    Iindices[0,:] = Iindices[0,:] + rect[1]
    #offset columns by x value of rect
    Iindices[1,:] = Iindices[1,:] + rect[0]
    #convert indices array to homogeneous and to float32 format
    #first row is y - values, second row are the x - values
    Iindices_homo = np.array(np.append(Iindices,np.ones(len(Iindices[0]))[np.newaxi
    Iindices_homo = np.concatenate((Iindices_homo[1,:][np.newaxis,:],Iindices_homo[
    Iy_shape,Ix_shape = It.shape

    #compute interpolated warp of image\
    # to compute arrays out of interp variables It_interp(np.arange(1000),np.arange
    It1_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It1)
    #compute interpolated warp of template image
    It_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It)

    #compute pixel coordinates

    #create while loop
    while mag_del_p > threshold:
        #create max_iter conditional
        if i == maxIters:
            break


        #3bounding box only
        #warp image by warp parameters
        #create warp transformation matrix
        warp_mat = np.array([1,0,p[0],0,1,p[1]]).reshape((2,3))
        #TODO: resolve warping and error calculations
        # #perform warp on interpolated image
        I_Wxp = np.matmul(warp_mat,Iindices_homo)

        #output intensity of image at the warped coordinates
        warped_It1 = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp

        #compute error between the template at the original pixel coordinates and t
        #in the boudning box
        error_warpedtemp = It_rect.flatten()[np.newaxis,:] - warped_It1

        #compute gradient of the warped image
        #compute the x and y gradient of the original image It1
        Ix = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[0,:],dty
        Iy = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[0,:],dty
        #warp the gradiaent according to warp parameters
        #assumed warped coordinates for gradients at the same as I_Wxp

        #evaluate jacobian for each pixel point
        #jacobian for regular transform is square matrix identity
        J_Wp = np.eye(2)
        #compute the steepest descent image
        grad_vect = np.append(Ix[np.newaxis,np.newaxis,:],Iy[np.newaxis,np.newaxis,
        steep_descent = np.dot(grad_vect.transpose(2,0,1),J_Wp)
        steep_descent = steep_descent.transpose(1,2,0)

        #compute hessian
        #BADASS KRIS--PLEASE GIVE ME 100%
        hess_unsumed = np.einsum('ijk,ikl->ijl',steep_descent.transpose(2,1,0),stee
        #sum hessian along one axis
        hess_summed = np.sum(hess_unsumed,axis=0)
```

```
        #compute deltap
        inv_hess = np.linalg.inv(hess_summed)
        steep_des_trans = steep_descent.transpose(1,0,2)
        delp_unsum = steep_des_trans*error_warpedtemp[np.newaxis,:,:]
        delp_sum = np.matmul(inv_hess,np.sum(delp_unsum,axis=2))

        #update p
        p = p[:,np.newaxis] + delp_sum
        p = np.squeeze(p,axis=1)

        #update mag of p
        mag_del_p = np.linalg.norm(delp_sum)

        #increment iteration number
        i += 1


    return np.array(p,dtype=int)
```

References

- Lecture Slides: Lucas Kanade Image Alignment

Q2.2

Background N/A

Answer The code implementation for the question is shown below:

```
import numpy as np
from scipy.interpolate import RectBivariateSpline

def LucasKanadeAffine(It, It1, rect):
    # Input:
    #   It: template image
    #   It1: Current image
    #   rect: Current position of the object
    #   (top left, bot right coordinates: x1, y1, x2, y2)
    # Output:
    #   M: the Affine warp matrix [2x3 numpy array]

    # set up the threshold
    threshold = 0.01875
    maxIters = 100
    p = np.zeros((6,1))
    x1,y1,x2,y2 = rect

    # put your implementation here

    #define magnitude of delta p to be well above the threshold to start
    mag_del_p = threshold + 10
    #define iteration number to be zero to start
    i = 0

    Iy_shape,Ix_shape = It.shape

    if x1 > x2:
        temp = x2
        x1 = x2
        x1 = temp

    if y1 > y2:
        temp = y2
        y2 = y1
        y1 = temp

    if x1 > Ix_shape:
        x1 = Ix_shape
    elif x1 < 0:
        x1 = 0

    if x2 > Ix_shape:
        x2 = Ix_shape
    elif x2 < 0:
        x2 = 0

    if y1 > Iy_shape:
        y1 = Iy_shape
    elif y1 < 0:
        y1 = 0

    if y2 > Iy_shape:
        y2 = Iy_shape
    elif y2 < 0:
        y2 = 0

    #compute interpolated warp of image\
    # to compute arrays out of interp variables It_interp(np.arange(1000),np.arange
    It1_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It1)
    #compute interpolated warp of template image
    It_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It)

    rect_y_range = np.linspace(np.int32(y1),np.int32(y2),num=np.int32(y2-y1))
    rect_x_range = np.linspace(np.int32(x1),np.int32(x2),num=np.int32(x2-x1))
    #It_rect = It[np.int32(y1):np.int32(y2)+1,np.int32(x1):np.int32(x2)+1]
    It_rect = It_interp(rect_y_range,rect_x_range)

    onesI_mat = np.ones_like(It_rect)
    Iindices = np.array(np.where(onesI_mat>=0))
    #offset rows by y value of rect
    Iindices[0,:] = Iindices[0,:] + y1
    #offset columns by x value of rect
    Iindices[1,:] = Iindices[1,:] + x1
    #convert indices array to homogeneous and to float32 format
    #first row is y - values, second row are the x - values
    Iindices_homo = np.array(np.append(Iindices,np.ones(len(Iindices[0]))[np.newaxi
    Iindices_homo = np.concatenate((Iindices_homo[1,:][np.newaxis,:],Iindices_homo[

    #initialize the

    #create while loop
    while mag_del_p > threshold:
        #create max_iter conditional
        if i == maxIters:
            break

        #warp image by warp parameters
        # reshape the output affine matrix
        warp_mat = np.array([[1.0+p[0], p[1],    p[2]],
                      [p[3],     1.0+p[4], p[5]]]).reshape(2, 3)
```

```
#perform warp on interpolated image
I_Wxp = np.matmul(warp_mat,Iindices_homo)

#output intensity of image at the warped coordinates
warped_It1 = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[

#compute error between the template at the original pixel coordinates and t
#in the boudning box
error_warpedtemp = It_rect.flatten()[np.newaxis,:] - warped_It1

#compute gradient of the warped image
#compute the x and y gradient of the original image It1
Ix = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[0,:],dty
Iy = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[0,:],dty
#warp the gradiaent according to warp parameters
#assumed warped coordinates for gradients at the same as I_Wxp

#evaluate jacobian for each pixel point
#create zero matrix
len_win = len(I_Wxp[0])
J_Wp = np.zeros((2,6,len_win))
#compute jacobian

J_Wp[0,0,:] = I_Wxp[0,:]
J_Wp[0,1,:] = I_Wxp[1,:]
J_Wp[0,2,:] = np.ones((len_win))
J_Wp[1,3,:] = I_Wxp[0,:]
J_Wp[1,4,:] = I_Wxp[1,:]
J_Wp[1,5,:] = np.ones((len_win))


#compute the steepest descent image
grad_vect = np.append(Ix[np.newaxis,np.newaxis,:],Iy[np.newaxis,np.newaxis,
steep_descent = np.einsum('ijk,ikl->ijl',grad_vect.transpose(2,0,1),J_Wp.tr
steep_descent = steep_descent.transpose(1,2,0)

#compute hessian
hess_unsumed = np.einsum('ijk,ikl->ijl',steep_descent.transpose(2,1,0),stee
#sum hessian along one axis
hess_summed = np.sum(hess_unsumed,axis=0)

#compute deltap
inv_hess = np.linalg.inv(hess_summed)
steep_des_trans = steep_descent.transpose(1,0,2)
delp_unsum = steep_des_trans*error_warpedtemp[np.newaxis,:,:]
delp_sum = np.matmul(inv_hess,np.sum(delp_unsum,axis=2))
```

```
        #update p
        p = p + delp_sum

        #update mag of p
        mag_del_p = np.linalg.norm(delp_sum)

        #increment iteration number
        i += 1

    M = warp_mat

    return M
```

References

- Lecture Slides: Lucas Kanade Image Alignment

Q2.3

Background N/A

Answer The code implementation for the question is shown below:

```
import numpy as np
from scipy.interpolate import RectBivariateSpline
import cv2

def InverseCompositionAffine(It, It1, rect):
    # Input:
    #   It: template image
    #   It1: Current image
    #   rect: Current position of the object
    #   (top left, bot right coordinates: x1, y1, x2, y2)
    # Output:
    #   M: the Affine warp matrix [2x3 numpy array]

    # set up the threshold
    threshold = 0.01875
    maxIters = 100
    p = np.zeros((6,1))
    x1,y1,x2,y2 = rect

    # put your implementation here

    #define magnitude of delta p to be well above the threshold to start
    mag_del_p = threshold + 10
    #define iteration number to be zero to start
    i = 0

    Iy_shape,Ix_shape = It.shape

    if x1 > x2:
        temp = x2
        x1 = x2
        x1 = temp

    if y1 > y2:
        temp = y2
        y2 = y1
        y1 = temp

    if x1 > Ix_shape:
        x1 = Ix_shape
    elif x1 < 0:
        x1 = 0
```

```
if x2 > Ix_shape:
    x2 = Ix_shape
elif x2 < 0:
    x2 = 0

if y1 > Iy_shape:
    y1 = Iy_shape
elif y1 < 0:
    y1 = 0

if y2 > Iy_shape:
    y2 = Iy_shape
elif y2 < 0:
    y2 = 0

#compute interpolated warp of image\
# to compute arrays out of interp variables It_interp(np.arange(1000),np.arange
It1_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It1)
#compute interpolated warp of template image
It_interp = RectBivariateSpline(np.arange(Iy_shape),np.arange(Ix_shape),It)

rect_y_range = np.linspace(np.int32(y1),np.int32(y2),num=200)
rect_x_range = np.linspace(np.int32(x1),np.int32(x2),num=200)
#It_rect = It[np.int32(y1):np.int32(y2)+1,np.int32(x1):np.int32(x2)+1]
It_rect = It_interp(rect_y_range,rect_x_range)

onesI_mat = np.ones_like(It_rect)
Iindices = np.array(np.where(onesI_mat>=0))
#offset rows by y value of rect
Iindices[0,:] = Iindices[0,:] + y1
#offset columns by x value of rect
Iindices[1,:] = Iindices[1,:] + x1
#convert indices array to homogeneous and to float32 format
#first row is y - values, second row are the x - values
Iindices_homo = np.array(np.append(Iindices,np.ones(len(Iindices[0]))[np.newaxi
Iindices_homo = np.concatenate((Iindices_homo[1,:][np.newaxis,:],Iindices_homo[

#compute gradient of the template
Ix_template = It_interp.ev(Iindices_homo[1,:],Iindices_homo[0,:],dy=1)
Iy_template = It_interp.ev(Iindices_homo[1,:],Iindices_homo[0,:],dx=1)

#evaluate jacobian at initial
len_win = len(Iindices_homo[0])
J_temp = np.zeros((2,6,len_win))
J_temp[0,0,:] = Iindices_homo[0,:]
J_temp[0,1,:] = Iindices_homo[1,:]
J_temp[0,2,:] = np.ones((len_win))
J_temp[1,3,:] = Iindices_homo[0,:]
```

```
J_temp[1,4,:] = Iindices_homo[1,:]
J_temp[1,5,:] = np.ones((len_win))

#compute steepest descent at initial
grad_template_vect = np.append(Ix_template[np.newaxis,np.newaxis,:],Iy_template
steep_descent = np.einsum('ijk,ikl->ijl',grad_template_vect.transpose(2,0,1),J_
steep_descent = steep_descent.transpose(1,2,0)



#compute hessian matrix at initial
hess_unsumed = np.einsum('ijk,ikl->ijl',steep_descent.transpose(2,1,0),steep_de
#sum hessian along one axis
hess_summed = np.sum(hess_unsumed,axis=0)
# hess_summed = steep_descent.squeeze() @ np.transpose(steep_descent.squeeze())
inv_hess = np.linalg.inv(hess_summed)

steep_des_trans = steep_descent.transpose(1,0,2)

#intialize delp_sum to be zeros
delp_sum = np.zeros((6,1))

#initialize warp
warp_mat = np.array([[1.0+p[0], p[1],    p[2]],
              [p[3],     1.0+p[4], p[5]]]).reshape(2, 3)

#create while loop
while mag_del_p > threshold:
    #create max_iter conditional
    if i == maxIters:
        break

    # #perform warp on interpolated image
    I_Wxp = np.matmul(warp_mat,Iindices_homo)

    #output intensity of image at the warped coordinates
    warped_It1 = It1_interp.ev(np.array(I_Wxp[1,:],dtype=float),np.array(I_Wxp[

    #compute error between the template at the original pixel coordinates and t
    #in the boudning box
    error_warpedtemp = warped_It1 - It_rect.flatten()[np.newaxis,:]

    #compute deltap
    delp_unsum = steep_des_trans*error_warpedtemp[np.newaxis,:,:]
    delp_sum = np.matmul(inv_hess,np.sum(delp_unsum,axis=2))

    #update mag of p
    mag_del_p = np.linalg.norm(delp_sum)
```

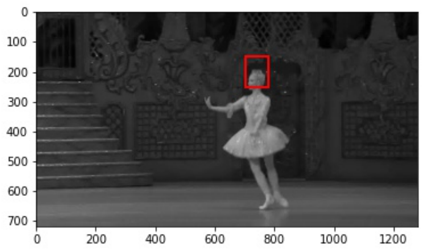
```
        warp_mat_delp = np.array([[1.0 + delp_sum[0], delp_sum[1],    delp_sum[2]],
                      [delp_sum[3],     1.0 + delp_sum[4], delp_sum[5]]]).reshape(2

        warp_mat = np.append(warp_mat,np.array([0,0,1])[np.newaxis,:],axis=0)

        warp_mat_delp_inv = cv2.invertAffineTransform(warp_mat_delp)

        warp_mat_delp_inv = np.append(warp_mat_delp_inv,np.array([0,0,1])[np.newaxi

        warp_mat = warp_mat @ warp_mat_delp_inv

        warp_mat = warp_mat[:2,:]

        #increment iteration number
        i += 1

    M = warp_mat[:2,:]

    return M
```

References

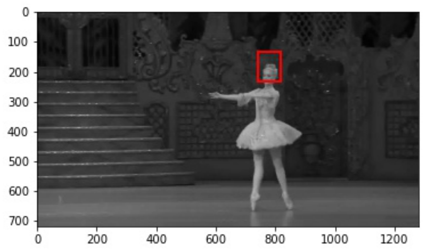
- Lecture Slides: Matthews Baker Image Alignment

Q2.4

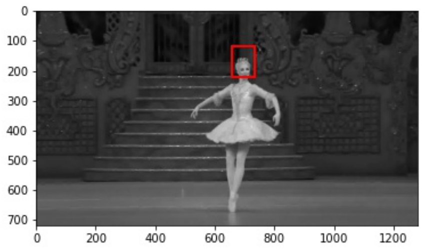
Translation-Only Lucas Kanade The figure shown below (see Figure 0.3) samples five frames of the output of the translation-only Lucas Kanade algorithm in the *ballet* video:



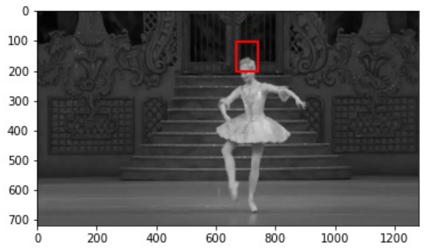
(a) Frame 01



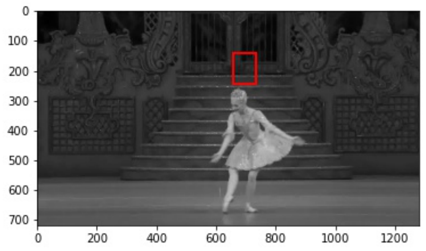
(b) Frame 05



(c) Frame 19



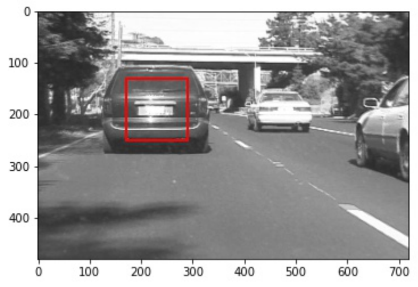
(d) Frame 34

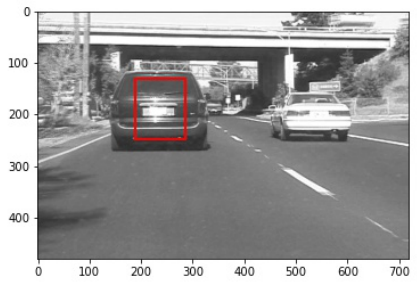


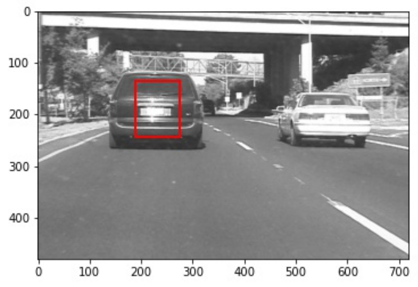
(e) Frame 41

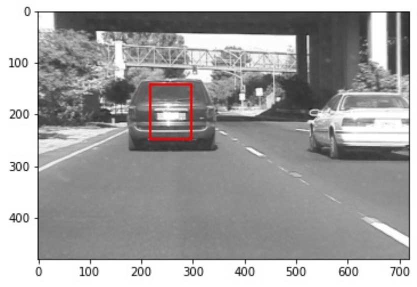
Figure 0.3: Translation-Only Lucas Kanade Tracking on Ballet Video

The figure shown below (see Figure 0.4) samples five frames of the output of the translation-only Lucas Kanade algorithm in the *car1* video:


(a) Frame 01


(b) Frame 51


(c) Frame 76


(d) Frame 115

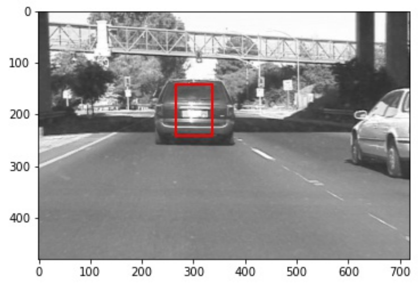
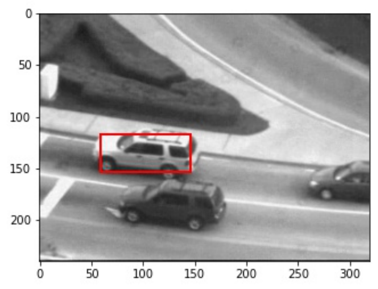
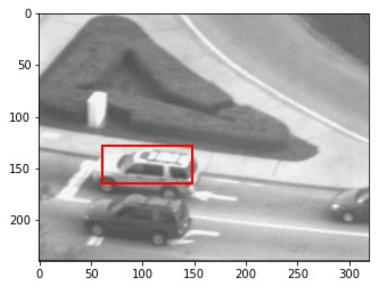

(e) Frame 154

Figure 0.4: Translation-Only Lucas Kanade Tracking on Car1 Video

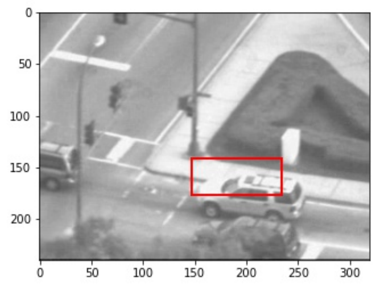
The figure shown below (see Figure 0.5) samples five frames of the output of the translation-only Lucas Kanade algorithm in the *car2* video:



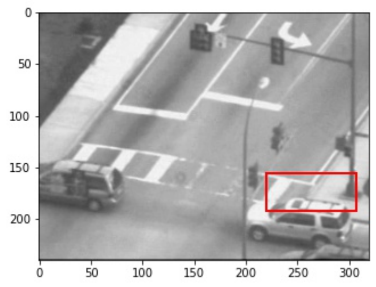
(a) Frame 01



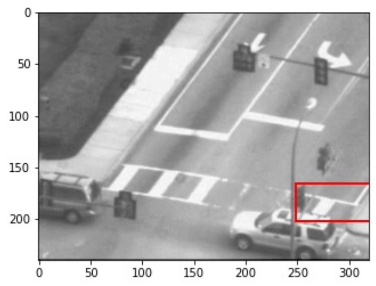
(b) Frame 23



(c) Frame 51



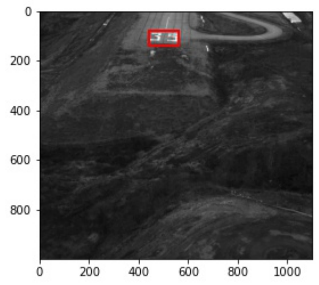
(d) Frame 79



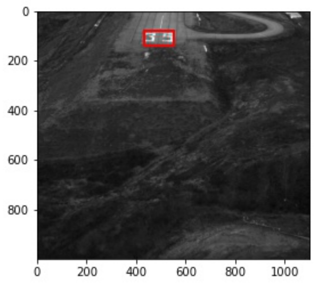
(e) Frame 93

Figure 0.5: Translation-Only Lucas Kanade Tracking on Car2 Video

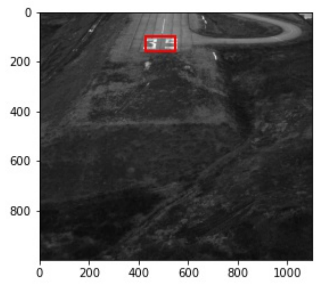
The figure shown below (see Figure 0.6) samples five frames of the output of the translation-only Lucas Kanade algorithm in the *landing* video:



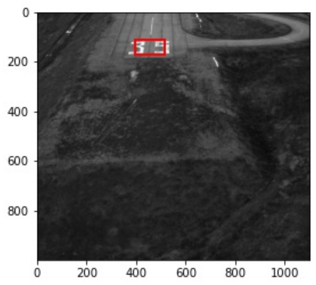
(a) Frame 01



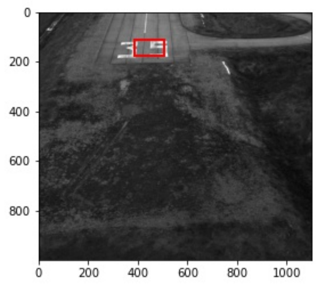
(b) Frame 10



(c) Frame 22



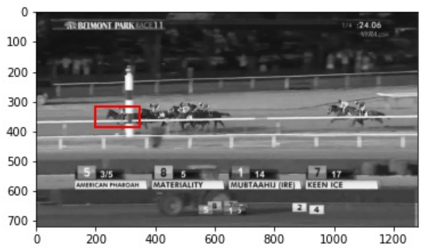
(d) Frame 38



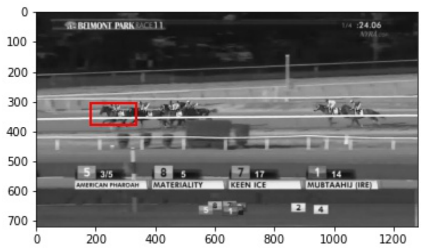
(e) Frame 47

Figure 0.6: Translation-Only Lucas Kanade Tracking on Landing Video

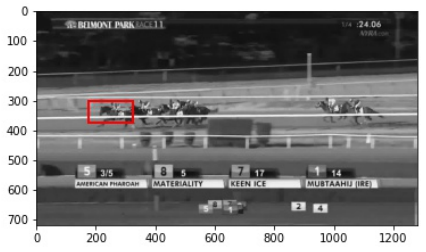
The figure shown below (see Figure 0.7) samples five frames of the output of the translation-only Lucas Kanade algorithm in the *race* video:



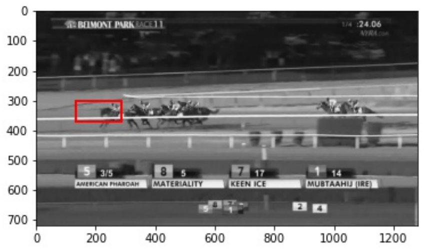
(a) Frame 01



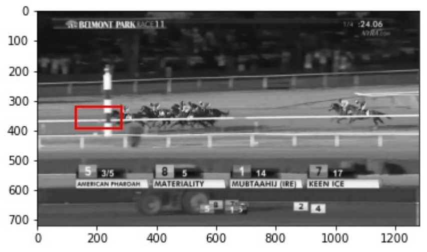
(b) Frame 22



(c) Frame 40



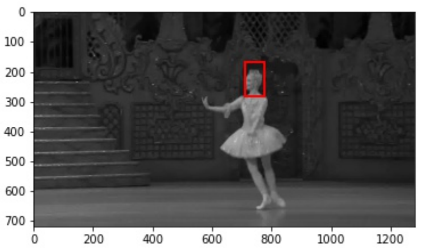
(d) Frame 54

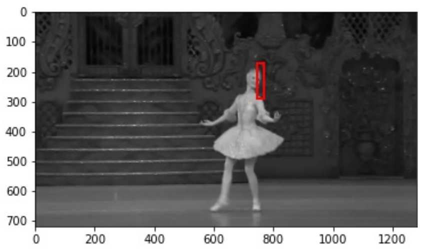


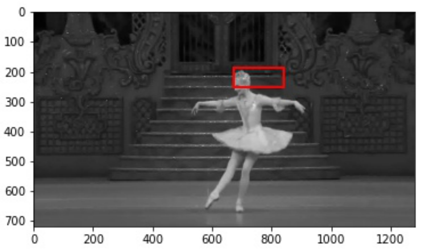
(e) Frame 67

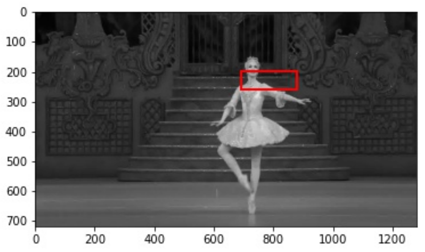
Figure 0.7: Translation-Only Lucas Kanade Tracking on Race Video

Affine Lucas Kanade The figure shown below (see Figure 0.8) samples five frames of the output of the affine Lucas Kanade algorithm in the *ballet* video:


(a) Frame 01


(b) Frame 12


(c) Frame 23


(d) Frame 32

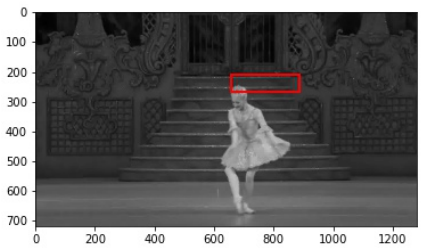
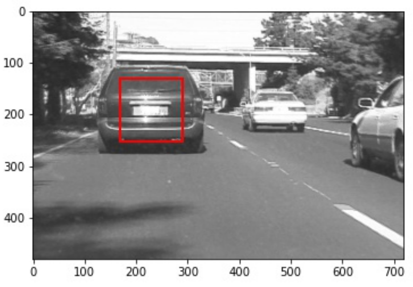
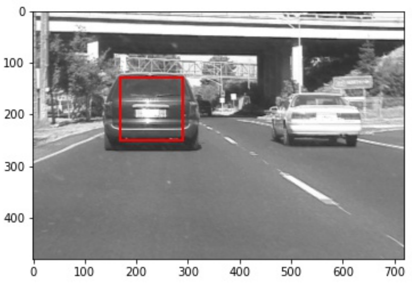

(e) Frame 42

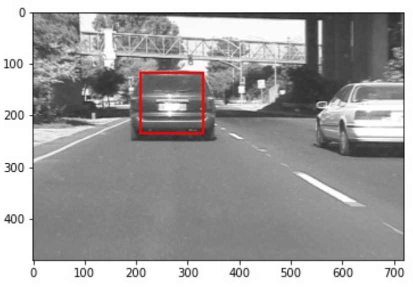
Figure 0.8: Affine Lucas Kanade Tracking on Ballet Video

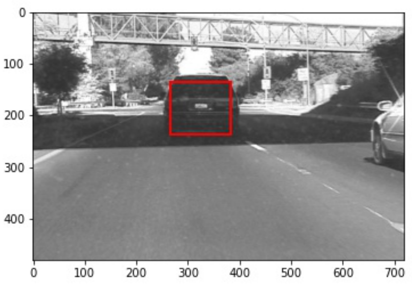
The computation of the figures shown above was performed using the *rect* variable to define the bounding box of the interpolated spline variable of the template image.

The figure shown below (see Figure 0.9) samples five frames of the output of the affine Lucas Kanade algorithm in the *car1* video:


(a) Frame 01


(b) Frame 71


(c) Frame 128


(d) Frame 172

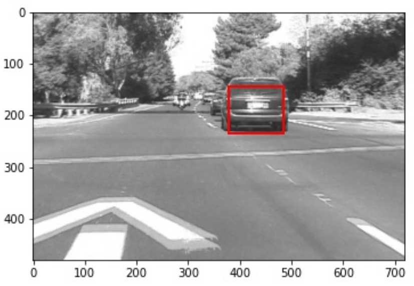
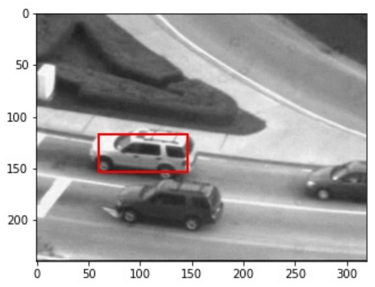
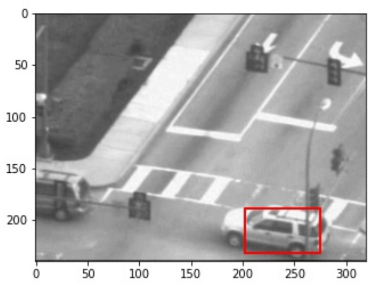

(e) Frame 245

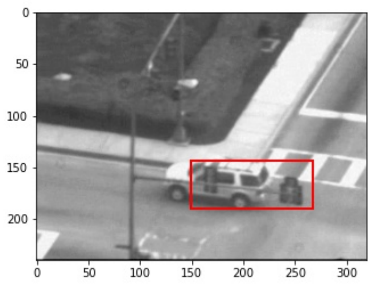
Figure 0.9: Affine Lucas Kanade Tracking on Car1 Video

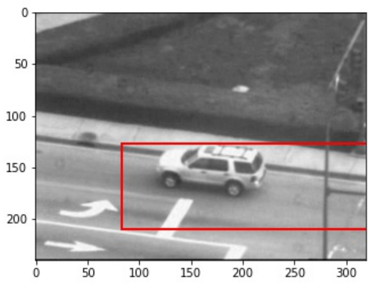
The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

The figure shown below (see Figure 0.10) samples five frames of the output of the affine Lucas Kanade algorithm in the *car2* video:


(a) Frame 01


(b) Frame 98


(c) Frame 140


(d) Frame 175

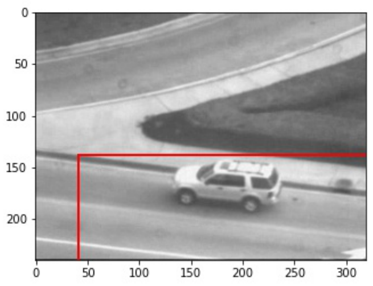
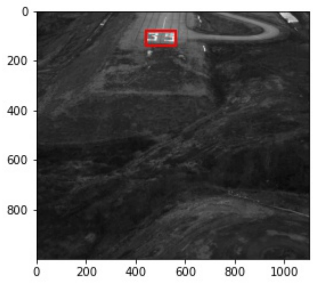
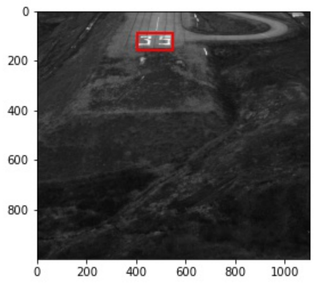

(e) Frame 214

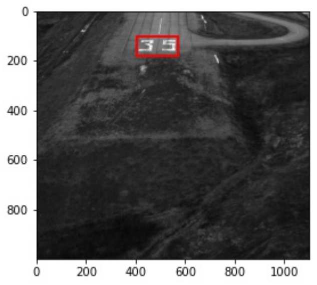
Figure 0.10: Affine Lucas Kanade Tracking on Car2 Video

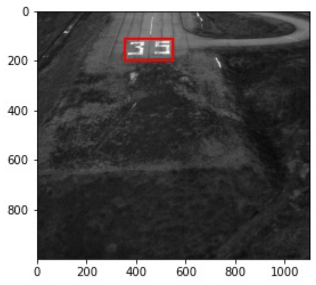
The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

The figure shown below (see Figure 0.11) samples five frames of the output of the affine Lucas Kanade algorithm in the *landing* video:


(a) Frame 01


(b) Frame 17


(c) Frame 29


(d) Frame 40

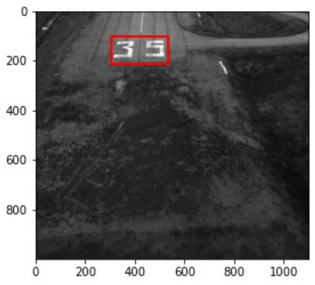
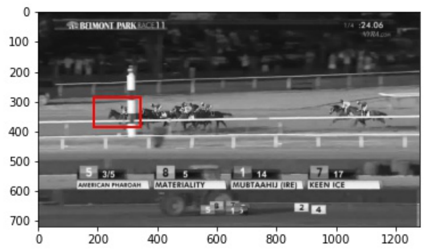

(e) Frame 48

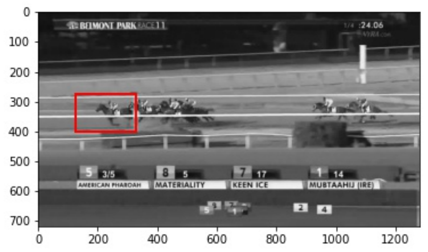
Figure 0.11: Affine Lucas Kanade Tracking on Landing Video

The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

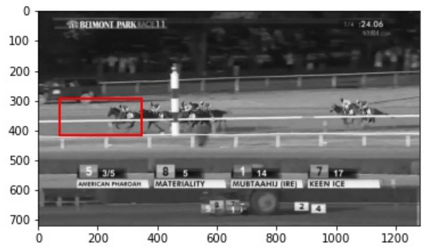
The figure shown below (see Figure 0.12) samples five frames of the output of the affine Lucas Kanade algorithm in the *race* video:



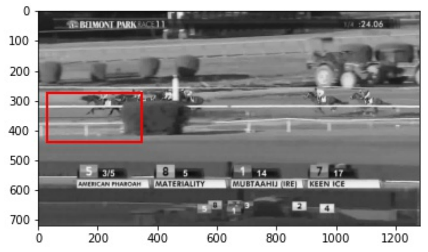
(a) Frame 01



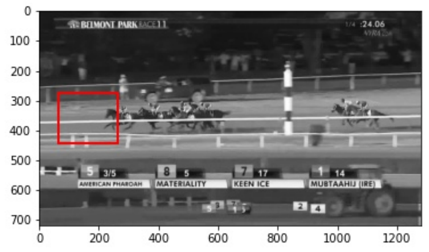
(b) Frame 08



(c) Frame 17



(d) Frame 27

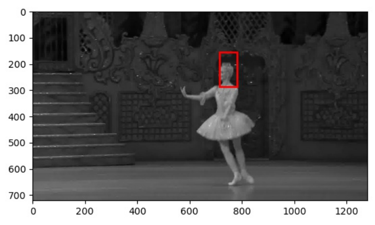


(e) Frame 35

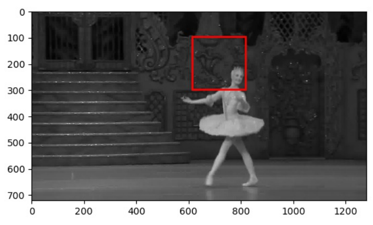
Figure 0.12: Affine Lucas Kanade Tracking on Race Video

The computation of the figures shown above was performed using the *rect* variable to define the bounding box of the interpolated spline variable of the template image.

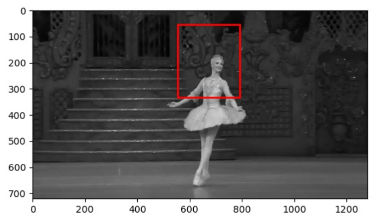
Matthew-Bakers Inverse Compositional The figure shown below (see Figure 0.13) samples five frames of the output of the Matthew-Bakers Inverse Compositional algorithm in the *ballet* video:



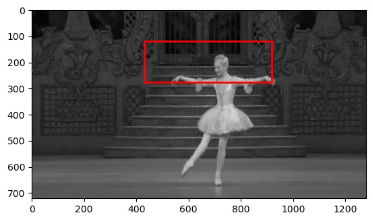
(a) Frame 01



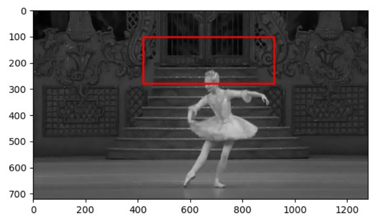
(b) Frame 08



(c) Frame 14



(d) Frame 25

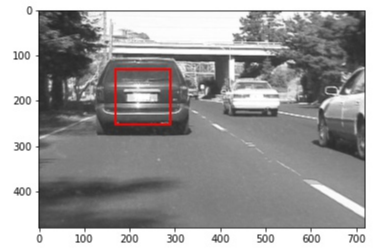


(e) Frame 36

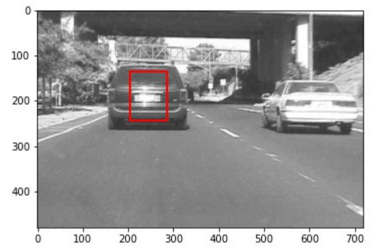
Figure 0.13: Inverse Compositional Tracking on Ballet Video

The computation of the figures shown above was performed using the *rect* variable to define the bounding box of the interpolated spline variable of the template image.

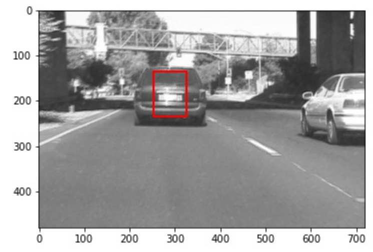
The figure shown below (see Figure 0.14) samples five frames of the output of the Matthew-Bakers Inverse Compositional algorithm in the *car1* video:



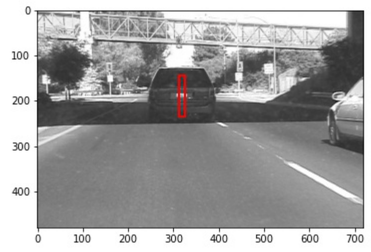
(a) Frame 01



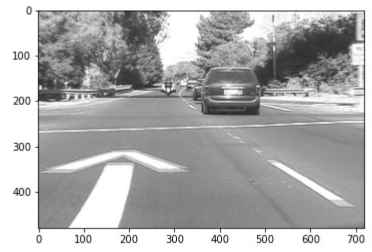
(b) Frame 98



(c) Frame 142



(d) Frame 167

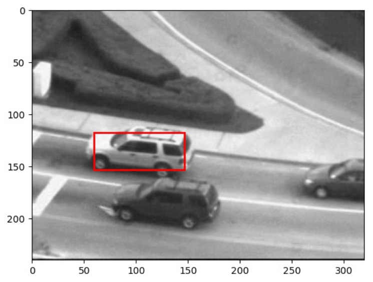


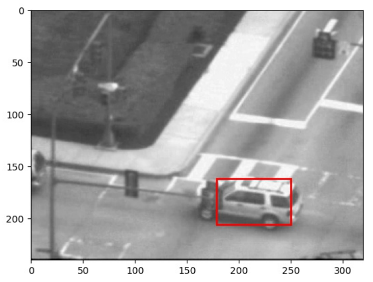
(e) Frame 240

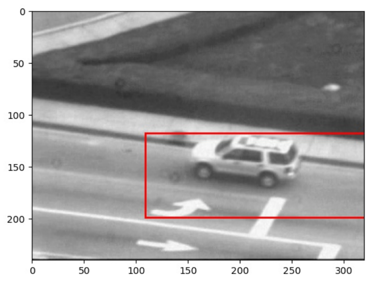
Figure 0.14: Inverse Compositional Tracking on Car1 Video

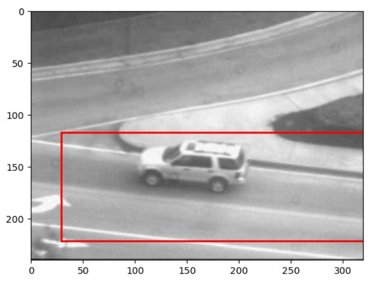
The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

The figure shown below (see Figure 0.15) samples five frames of the output of the Matthew-Bakers Inverse Compositional algorithm in the *car2* video:


(a) Frame 01


(b) Frame 119


(c) Frame 185


(d) Frame 239

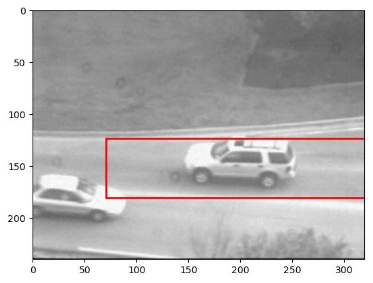
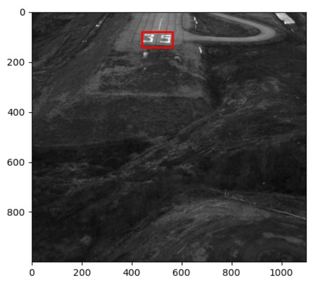

(e) Frame 300

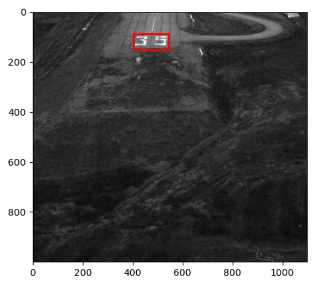
Figure 0.15: Inverse Compositional Tracking on Car2 Video

The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

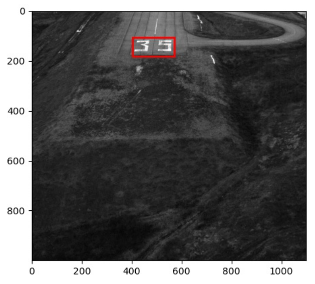
The figure shown below (see Figure 0.16) samples five frames of the output of the Matthew-Bakers Inverse Compositional algorithm in the *landing* video:



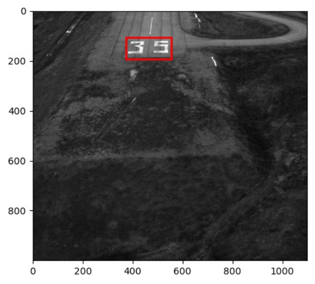
(a) Frame 01



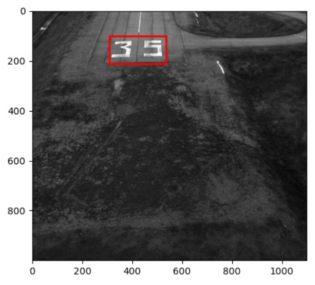
(b) Frame 15



(c) Frame 29



(d) Frame 36

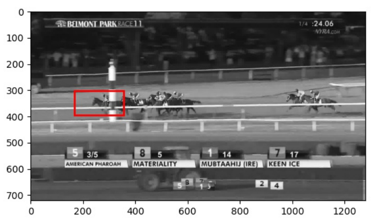


(e) Frame 47

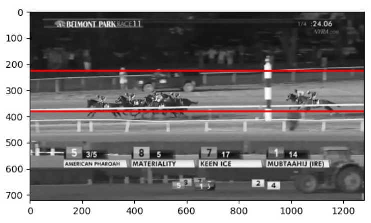
Figure 0.16: Inverse Compositional Tracking on Landing Video

The computation shown above was performed using the *rect* variable to define the bounding box of the *It* template image.

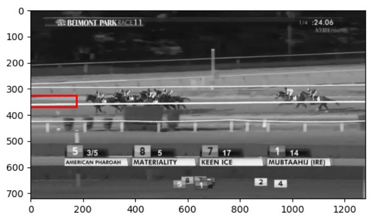
The figure shown below (see Figure 0.17) samples five frames of the output of the Matthew-Bakers Inverse Compositional algorithm in the *race* video:



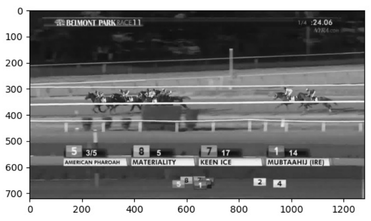
(a) Frame 01



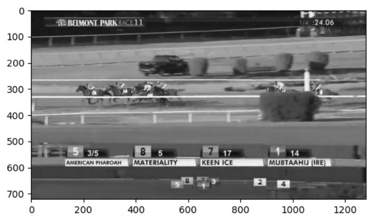
(b) Frame 18



(c) Frame 32



(d) Frame 55



(e) Frame 64

Figure 0.17: Inverse Compositional Tracking on Race Video

The computation of the figures shown above was performed using the *rect* variable to define the bounding box of the interpolated spline variable of the template image.

Analysis The main difference between the translation-only Lucas Kanade and affine Lucas Kanade algorithms is that the former has fewer parameters to update between two frames, and the latter utilizes a Jacobian matrix that will differ between pixel to pixel. This means that the Jacobian of the affine Lucas Kanade may be more prone to high frequency noise spanned across various pixels that will downplay the performance of the affine algorithm. The inverse compositional algorithm performs very similarly to the affine Lucas Kanade, and so can suffer from the same effects in some of the videos. The inverse compositional algorithm also has the benefit of being more computationally efficient compared to its affine counterpart.

For the *ballet* video, the translation-only Lucas Kanade algorithm performed the best. Though the bounding box could not track the ballerina when she lowered the level of her head significantly towards the end of the video, the bounding box was still able to track her head well before then. The affine Lucas Kanade and inverse compositional algorithms performed more poorly in accurately tracking the ballerina's head. The poor performance of the ballerina begins at the early frames of the video: when the ballerina turns her head, the bounding box can be seen collapsing on itself as it tries to track the ballerina's face part of the head. Afterwards, the error accumulated was significantly and caused the bounding box to widen further.

For the *car1* video, the translational and affine Lucas Kanade algorithms performed well to track the car, even when it performs the lane shift in the middle of the video. The inverse compositional algorithm, however, did not perform very well. When the car made the began the lane shift, the bounding box on the car began to collapse in on itself.

For the *car2* video, all three algorithms performed the same and suffered the same issue. When the car passing the intersection, the algorithms start to expanding the bounding box and track the stoplight that the car passes through from that camera angle. This can be attributed, for all the algorithms, and the accumulation of error prior and the fact that once the algorithm begins to track, it cannot resolve itself to look past the stop light.

For the *landing* video, the affine Lucas Kanade and inverse compositional algorithms performed well. The sizing warp parameters performed well to resize the bounding box in according to the rate at which the landing was growing as the airplane was traveling closer to the runway. The translation-only Lucas Kanade algorithm, however, did not perform so well as the size of the landing in the video was growing larger and larger. Because the algorithm does not have any parameters to change the size of the bounding box, the only thing that the algorithm can do is translate itself. Towards the end of the video, the bounding box was positioning itself at the upper right corner of the landing.

For the *race* video, all three algorithms performed the worst compared to the other videos for analysis. Though each algorithm had some period of time where their bounding boxes were encompassing the bulk of the body of the horse, over time the bounding box drifting away and expanded itself. This can be attributed to the fact that the horses were occasionally blocked partially due to bushes in the way or the wooden fencing.