Computer Vision
16720-B Fall 2021

16720 (B) Object Tracking in Videos - Assignment 6

Instructor: Kris TAs: Wen-Hsuan (Lead), Zen, Yan, Rawal, Paritosh, Qichen

Instructions

This section should include the visualizations and answers to specifically highlighted questions from Q1 to Q3. This section will need to be uploaded to gradescope as a pdf and manually graded (this is a separate submission from the coding notebooks)

1. Students are encouraged to work in groups but each student must submit their own work. Include the names of your collaborators in your write up. Code should Not be shared or copied. Please properly give credits to others by LISTING EVERY COLLABORATOR in the writeup including any code segments that you discussed, Please DO NOT use external code unless permitted. Plagiarism is prohibited and may lead to failure of this course.
2. **Start early!** This homework will take a long time to complete.
3. **Questions:** If you have any question, please look at Piazza first and the FAQ page for this homework.
4. All the theory question and manually graded questions should be included in a single writeup (this notebook exported as pdf or a standalone pdf file) and submitted to gradescope: pdf assignment.
5. **Attempt to verify your implementation as you proceed:** If you don't verify that your implementation is correct on toy examples, you will risk having a huge issue when you put everything together. We provide some simple checks in the notebook cells, but make sure you verify them on more complicated samples before moving forward.
6. **Do not import external functions/packages other than the ones already imported in the files:** The current imported functions and packages are enough for you to complete this assignment. If you need to import other functions, please remember to comment them out after submission. Our autograder will crash if you import a new function that the gradescope server does not expect.
7. Assignments that do not follow this submission rule will be **penalized up to 10% of the total score**.

Preliminaries

In this section, we will go through some of the basics of the Lucas-Kanade tracker and the Matthews-Baker tracker. The following table contains a summary of the variables used in the rest of the assignment.

Symbol	Vector/Matrix Size	Description
u	1×1	Image horizontal coordinate
v	1×1	Image vertical coordinate
$\mathbf{x}$	2×1 or 1×1	pixel coordinates: (u, v) or unrolled
$\mathbf{I}$	$m \times 1$	Image unrolled into a vector (m pixels)
$\mathbf{T}$	$m \times 1$	Template unrolled into a vector (m pixels)
$\mathbf{W}(\mathbf{p})$	3×3	Affine warp matrix
$\mathbf{p}$	6×1	parameters of affine warp
$\frac{\partial \mathbf{I}}{\partial u}$	$m \times 1$	partial derivative of image wrt u
$\frac{\partial \mathbf{I}}{\partial v}$	$m \times 1$	partial derivative of image wrt v
$\frac{\partial \mathbf{T}}{\partial u}$	$m \times 1$	partial derivative of template wrt u
$\frac{\partial \mathbf{T}}{\partial v}$	$m \times 1$	partial derivative of template wrt v
$\nabla \mathbf{I}$	$m \times 2$	image gradient $\nabla \mathbf{I}(\mathbf{x}) = \left[\frac{\partial \mathbf{I}(\mathbf{x})}{\partial u} \frac{\partial \mathbf{I}(\mathbf{x})}{\partial v}\right]$
$\nabla \mathbf{T}$	$m \times 2$	image gradient $\nabla \mathbf{T}(\mathbf{x}) = \left[\frac{\partial \mathbf{T}(\mathbf{x})}{\partial u} \frac{\partial \mathbf{T}(\mathbf{x})}{\partial v}\right]$
$\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$	2×6	Derivative of affine warp wrt its parameters
$\mathbf{J}$	$m \times 6$	$\nabla \mathbf{I} \frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ or $\nabla \mathbf{T} \frac{\partial \mathbf{W}}{\partial \mathbf{p}}$
$\mathbf{H}$	6×6	$\mathbf{J}^T \mathbf{J}$

Template

A template describes the object of interest (eg. a car, football) which we wish to track in a video. Traditionally, the tracking algorithm is initialized with a template, which is represented by a bounding box around the object to be tracked in the first frame of the video. For each of the subsequent frames in the video, the tracker will update its estimate of the object in the image. The tracker achieves this by updating its affine warp.

Warps

What is a warp? An image transformation or warp $\mathbf{W}$ is a function that acts on pixel coordinates $\mathbf{x} = [u \;\; v]^T$ and maps pixel values from one place to another in an image $\mathbf{x}' = \begin{bmatrix} u' & v' \end{bmatrix}^T$. Simply put, $\mathbf{W}$ maps a pixel with coordinates $\mathbf{x} = [u \;\; v]^T$ to $\mathbf{x}' = \begin{bmatrix} u' & v' \end{bmatrix}^T$. Translation, rotation, and scaling are all examples of warps. We denote the parameters of the warp function $\mathbf{W}$ by $\mathbf{p}$:

$$\mathbf{x}' = \mathbf{W}(\mathbf{x}; \mathbf{p})$$

Affine Warp

An affine warp is a particular kind of warp that can include any combination of translation, scaling, and rotations. An affine warp can be represented by 6 parameters $\mathbf{p} = [p_1 \;\; p_2 \;\; p_3 \;\; p_4 \;\; p_5 \;\; p_6]^T$. One of the most convenient things about an affine warp is that it is linear; its action on a point with

coordinates $\mathbf{x} = [u \quad v]^T$ can be described as a matrix operation by a 3×3 matrix $\mathbf{W}(\mathbf{p})$:,

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \mathbf{W}(\mathbf{p}) \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

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

Note: For convenience, when we want to refer to the warp as a function, we will use $\mathbf{W}(\mathbf{x}; \mathbf{p})$ and when we want to refer to the matrix for an affine warp, we will use $\mathbf{W}(\mathbf{p})$. We will use affine warp and affine transformation interchangeably.

Theory Questions (30 pts)

Before implementing the trackers, let's study some simple problems that will be useful during the implementation first. The answers to the below questions should be relatively short, consisting of a few lines of math and text.

Q1.1

Assuming the affine warp model defined above, derive the expression for the $\frac{\partial \mathbf{W}}{\partial \mathbf{p}}$ in terms of the warp parameters $\mathbf{p} = [p_1 \ p_2 \ p_3 \ p_4 \ p_5 \ p_6]'$.

Given:

$$W(P) = \begin{bmatrix} 1+P_1 & P_3 & P_5 \\ P_2 & 1+P_4 & P_6 \\ 0 & 0 & 1 \end{bmatrix}_{3\times 3}$$

$$\frac{\partial W}{\partial P} = \begin{bmatrix} \frac{\partial W^1(x,P)}{\partial P_1} & \frac{\partial W^1(x,P)}{\partial P_2} & \cdots & \frac{\partial W^1(x,P)}{\partial P_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial W^3(x,P)}{\partial P_1} & \frac{\partial W^3(x,P)}{\partial P_2} & & \frac{\partial W^3(x,P)}{\partial P_n} \end{bmatrix}$$

$$W(x,P) = \begin{bmatrix} 1+P_1 & P_3 & P_5 \\ P_2 & 1+P_4 & P_6 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} (1+P_1)x + P_3 y + P_5 \\ P_2 x + (1+P_4)y + P_6 \\ 1 \end{bmatrix}$$

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

Q1.2.

$$\frac{\partial W}{\partial p} = \begin{bmatrix} \frac{\partial W_1(x,p)}{\partial p_1} & \frac{\partial W_1(x,p)}{\partial p_2} & \frac{\partial W_1(x,p)}{\partial p_3} \\ \vdots & \vdots & \vdots \\ \frac{\partial W_2(x,p)}{\partial p_1} & \frac{\partial W_2(x,p)}{\partial p_2} & \frac{\partial W_2(x,p)}{\partial p_3} \end{bmatrix}$$

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

Q1.2

Find the computational complexity (Big O notation) for each runtime iteration (computing $\mathbf{J}$ and $\mathbf{H}^{-1}$) of the Lucas Kanade method. Express your answers in terms of n, m and p where n is the number of pixels in thetemplate $\mathbf{T}$, m is the number of pixels in an input image $\mathbf{I}$ and p is the number of parameters used to describe the warp W.

You may refer to the supplementary PDF for more detailed descriptions of the algorithm.

1.2. $n \rightarrow$ no: of pixels in Template T.

$p \rightarrow$ no: of warp parameters

$m \rightarrow$ no: of pixels in image I

Step 1: Warp I with $W(x;p)$ to compute $I(W(x,p))$

complexity $= O(np)$

(n pixels in T & p warp parameters)

Step 2: Compute error image

$$T(x) - I(W(x,p))$$

Complexity $= O(n)$ [n pixels in T]

Step 3: Warp the gradient ∇I with $W(x;p)$

complexity $= O(np)$ [same as step 1]

Step 4: Evaluate the Jacobian $\frac{\partial W}{\partial p}$ at $W(x,p)$

Complexity = $O(np)$ [depends on W but $O(p)$ per pixel]

Step5: Compute the steepest descent images

$$\underbrace{\nabla I}_{n\times 2} \; \underbrace{\frac{\partial W}{\partial p}}_{2\times p}$$

Complexity = $O(2np) = O(np)$

[matrix multiplication]

Step6: Computing Hessian from Jacobian.

$$H = \sum \underbrace{J^T}_{p\times n} \underbrace{J}_{n\times p} \quad \text{where} \quad J = \underbrace{\nabla I \frac{\partial W}{\partial p}}_{n\times p}$$

Complexity = $O(np^2)$.

Step 7: Compute $\sum \underbrace{\left[\nabla I \frac{\partial W}{\partial p}\right]^T}_{p\times n} \underbrace{\left[T(x) - I(W(x;p))\right]}_{n\times 1}$

complexity = $O(pn)$ → [matrix multiplication]

Step 8: Compute Δp using equation

$$\Delta p = H^{-1} \sum \underbrace{\left[\nabla I \frac{\partial W}{\partial p} \right]^T}_{p \times n} \underbrace{\left[T(x) - I(W(x,p)) \right]}_{n \times 1}$$

Complexity for $\underset{(p \times p)}{H^{-1}} = O(p^3)$

[polynomial form]

complexity of multiplication

$= O(p^2 n)$

$\therefore$ Overall complexity $= O(p^3)$

Step 9: Updating p

Overall complexity $= O(p)$

Overall for all the step,

Complexity $= O(p^3) + O(p^2 n)$

Q1.3

Find the computational complexity (Big O notation) for the initialization step (Precomputing $\mathbf{J}$ and $\mathbf{H}^{-1}$) and for each runtime iteration of the Matthews-Baker method. Express your answers in terms of n, m and p where n is the number of pixels in the template $\mathbf{T}$, m is the number of pixels in an input image $\mathbf{I}$ and p is the number of parameters used to describe the warp W. You may refer to the supplementary PDF for more detailed descriptions of the algorithm.

How does this compare to the run time of the regular Lucas-Kanade method?

1.3. $n \rightarrow$ no: of pixels in Template

$p \rightarrow$ no: of warp parameters

$m \rightarrow$ no: of pixels in Image.

Precomputation

Step 3: Evaluate the gradient ∇T of the template $T(x)$.

Complexity $\Rightarrow O(n)$. [Only pixels in Template]

Step 4: Evaluate the Jacobian $\frac{\partial W}{\partial p}$ at $(x,0)$

Complexity $= O(pn)$.

[n pixels in template & p warp parameters]

Step 5: Compute the steepest descent images.

$$\underset{n\times 2}{\nabla T} \; \underset{2\times p}{\frac{\partial W}{\partial p}}$$

Complexity $= O(pn)$.

Step 6: Compute the Hessian matrix

$$H = \sum \underbrace{\left[\nabla T \frac{\partial W}{\partial p}\right]^T}_{p \times n} \underbrace{\left[\nabla T \frac{\partial W}{\partial p}\right]}_{n \times p}$$

Complexity $= p^2 n$.

~~Step 7~~:

Iterate:

Step (1) Warp I with $W(x;p)$ to compute $I(W(x;p))$

Complexity $= O(np)$ [n pixels in Template warping with p parameters & sampling in I]

Step (2) Compute error $I(W(x,p)) - T(x)$.

Complexity $= O(n)$.

Step 7: Compute $\sum_x \left[\nabla T \frac{\partial W}{\partial p}\right]^T \left[I(W(x,p)) - T(x)\right]$

Complexity = (np).

Step 8: Compute Δp using

$$\Delta p = H^{-1} \sum \left[\nabla T \frac{\partial W}{\partial p} \right]^T \left[I(W(x;p) - T(x) \right]$$

Complexity = $O(p^2)$

[H^{-1} calculated outside]

Step 9: Update warp.

$$W(x;p) \leftarrow W(x,p) \circ W(x,\Delta p)^{-1}$$

Complexity = $O(p^2)$.

Overall complexity = $O(p^2 + pn)$.

Comparision with Lucas Kanade. → Since pre computation happens only once, in the mathew Bakers, the computation is much lesser than Lucas Kanade.

Also, From ~~from~~ the complexity analysis,

its visible That

for Lucas Kannad = $O(p^3+p^2n)$

For Mathew Babers = $O(p^2+pn)$

Hence, mathew babers is much faster.

Coding Questions Write-up

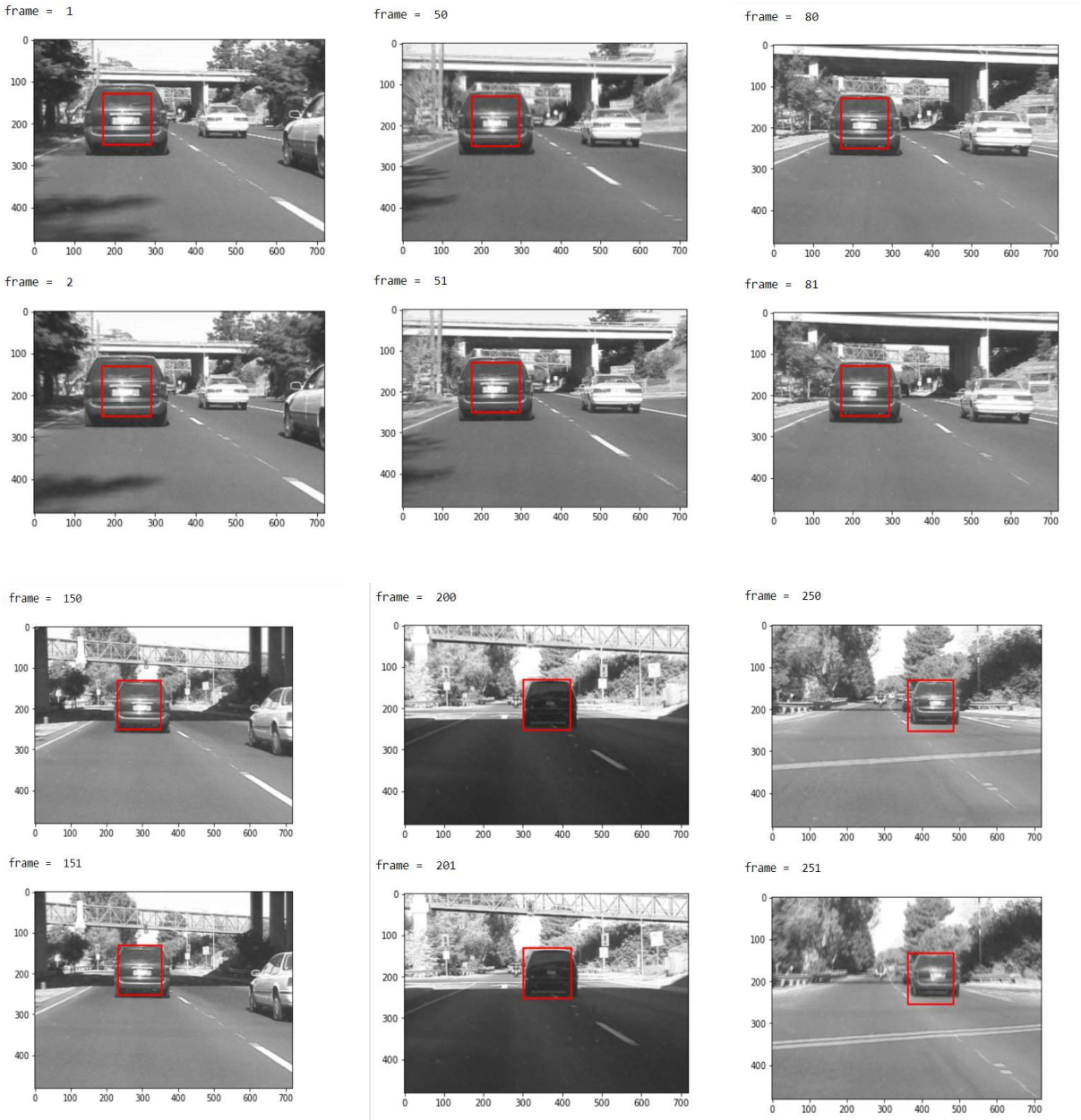
Q1.1

Code

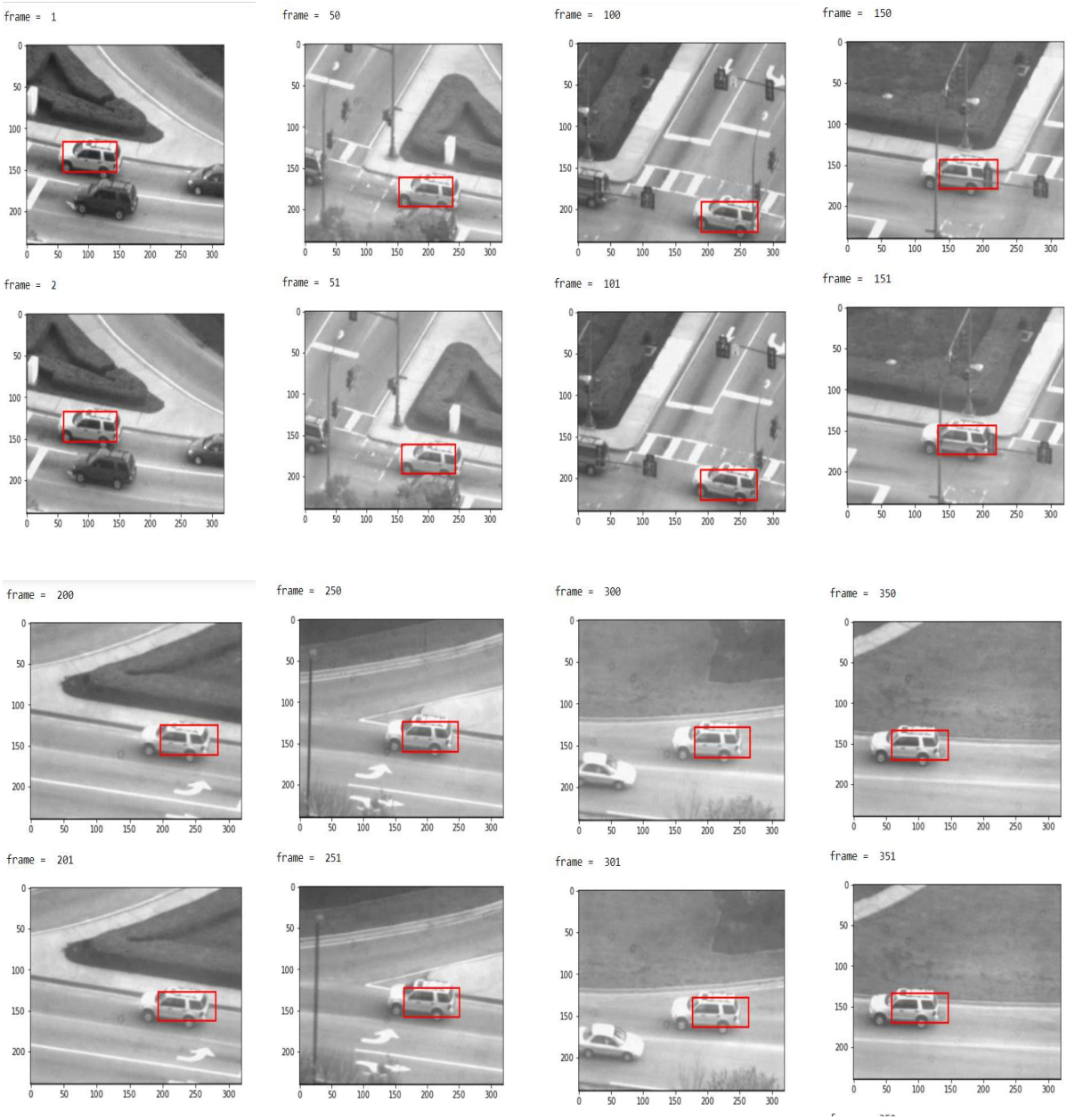
In [24]:
```
def LucasKanade(It, It1, rect, thresh=.025, maxIt=100):

    '''
    Q1.1: Lucas-Kanade Forward Additive Alignment with Translation Only

      Inputs:
        It: template image
        It1: Current image
        rect: Current position of the object
        (top left, bottom right coordinates, x1, y1, x2, y2)
        thresh: Stop condition when dp is too small
        maxIt: Maximum number of iterations to run

      Outputs:
        p: movement vector dx, dy
    '''

    # Set thresholds (you probably want to play around with the values)
    p = np.zeros(2) # dx, dy
    threshold = thresh
    maxIters = maxIt
    i = 0
    deltap = np.array([np.finfo('float').max, np.finfo('float').max])
    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3]
    It_int = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]), It)
    It1_int = RectBivariateSpline(np.arange(It1.shape[0]), np.arange(It1.shape[1]), It1)

    xt = np.arange(x1, x2 + 0.5)
    yt = np.arange(y1 , y2 + 0.5)
    xg,yg = np.meshgrid(xt, yt)
    #print(xg.shape)
    #print(yg.shape)
    template = It_int.ev(yg, xg)

    h, w = It1.shape
    x = np.arange(w)
    y = np.arange(h)
    while (np.linalg.norm(deltap) > threshold and i < maxIt):
```

```
        xti = np.arange(x1 + p[0], x2 + p[0] + 0.5)
        yti = np.arange(y1 + p[1], y2 + p[1] + 0.5)
        xgi,ygi = np.meshgrid(xti, yti)
        #print(xgi.shape)
        #print(ygi.shape)
        ind = (xgi > 0) & (ygi > 0 ) & (xgi < It1.shape[1]) & (ygi < It1.shape[0])
        #print(ind)
        #print(ygi.shape)
        xgi = xgi[ind]
        ygi = ygi[ind]

        patch = It1_int.ev(ygi, xgi)
        b = (template[ind] - patch).flatten()
        #print(b.shape)
        Ix = It1_int.ev(ygi, xgi, dx = 0, dy = 1)
        Iy = It1_int.ev(ygi, xgi, dx = 1, dy = 0)

        Iy = Iy.flatten()
        Ix = Ix.flatten()
        gradI = np.array([Ix,Iy]).T
        #print(gradI.shape)
        deltap = np.linalg.lstsq(gradI,b,rcond=None)[0]
        p = p + deltap
        #print(deltap)
        #print(p)
        i+=1
    #raise NotImplementedError()


    return p
```

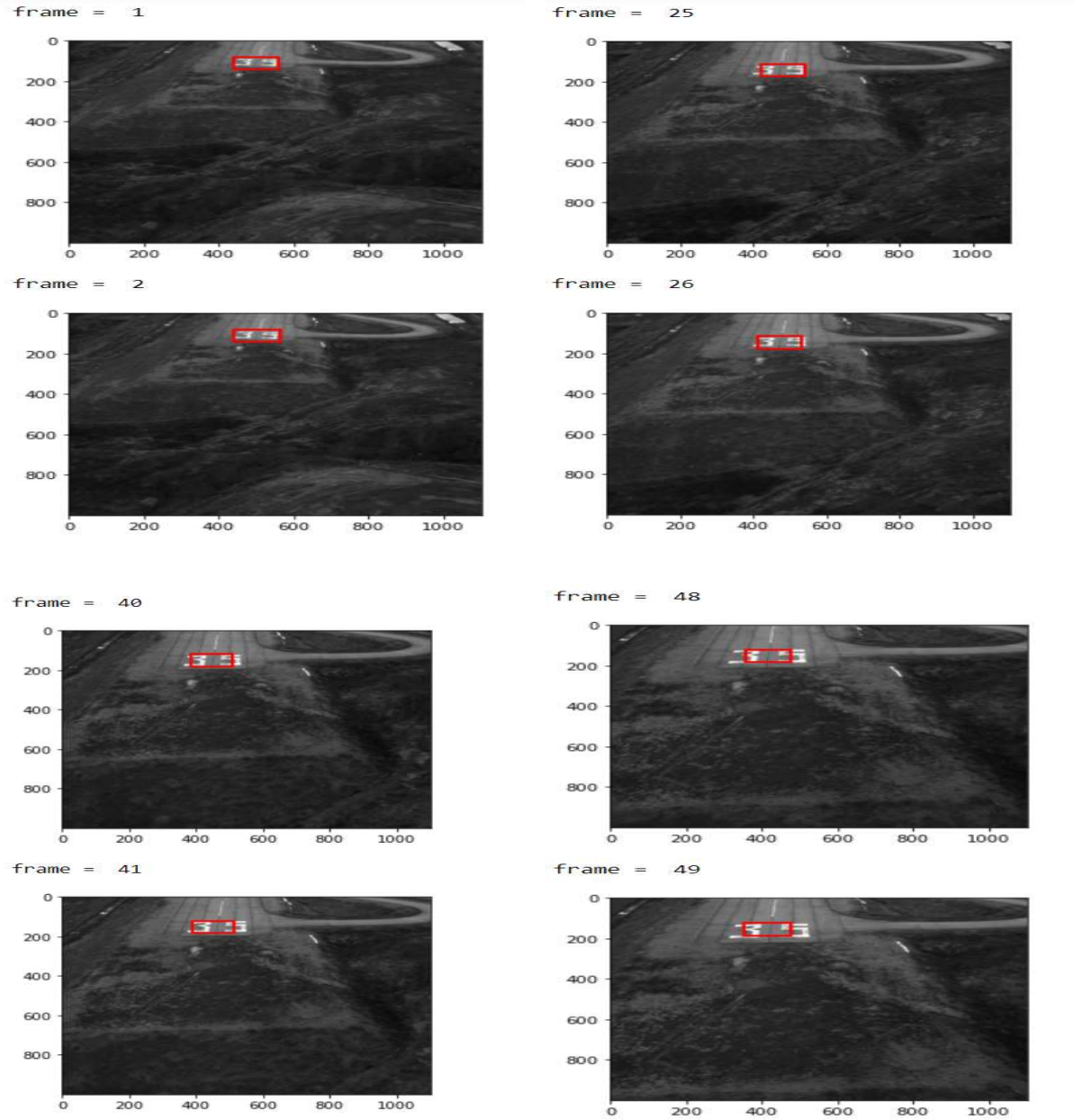
1) Car 1


frame = 1
frame = 50
frame = 80
frame = 2
frame = 51
frame = 81
frame = 150
frame = 200
frame = 250
frame = 151
frame = 201
frame = 251

2) Car 2

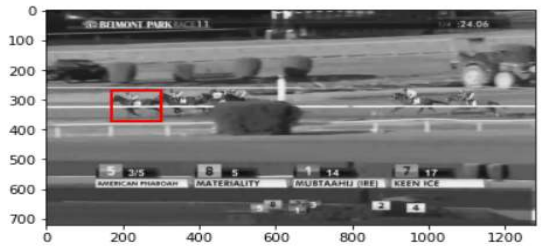

frame = 1
frame = 50
frame = 100
frame = 150
frame = 2
frame = 51
frame = 101
frame = 151
frame = 200
frame = 250
frame = 300
frame = 350
frame = 201
frame = 251
frame = 301
frame = 351

3) Landing


frame = 1
frame = 25
frame = 2
frame = 26
frame = 40
frame = 48
frame = 41
frame = 49

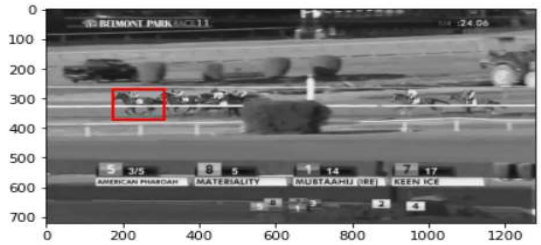
4) Racing

```
frame =  1
```

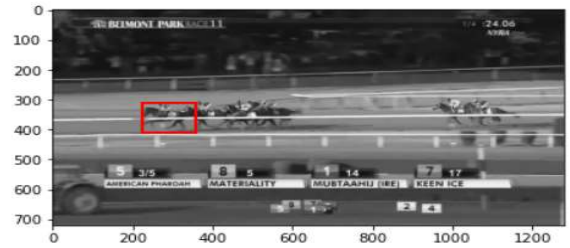


```
frame =  50
```

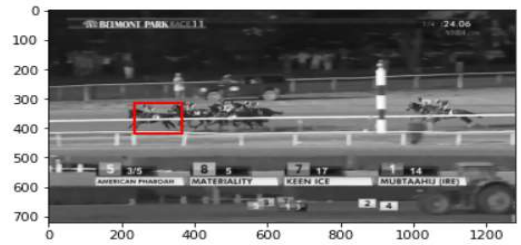
```
frame =  2
```



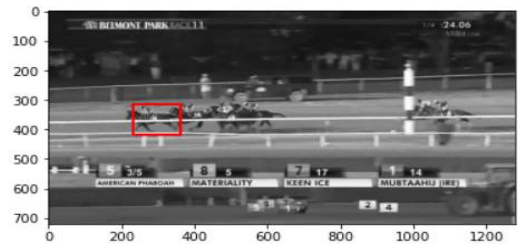
```
frame =  51
```



```
frame =  66
```

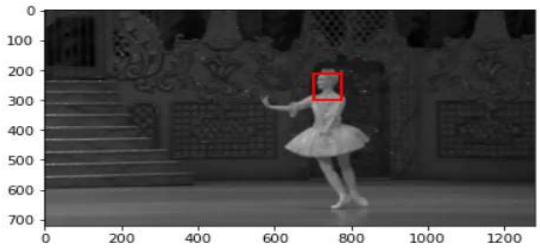


```
frame =  67
```

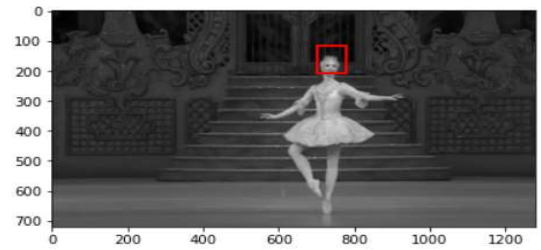


5) Ballet

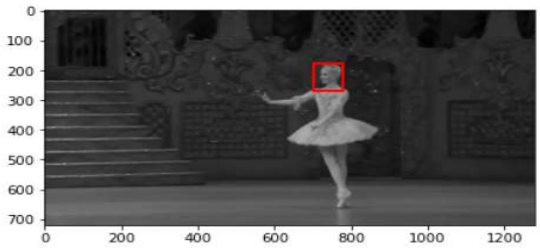
frame = 1



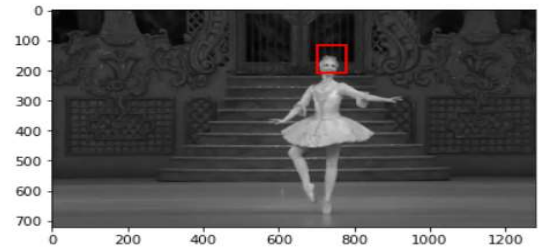
frame = 30



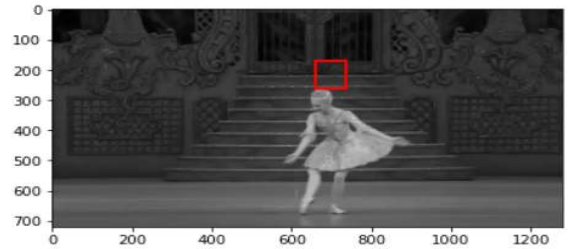
frame = 2



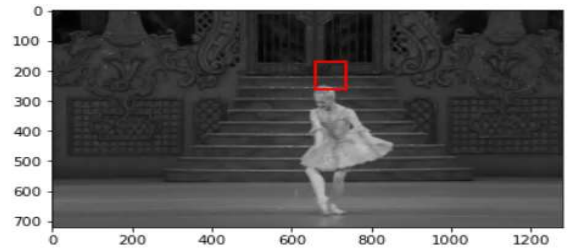
frame = 31



frame = 41



frame = 42



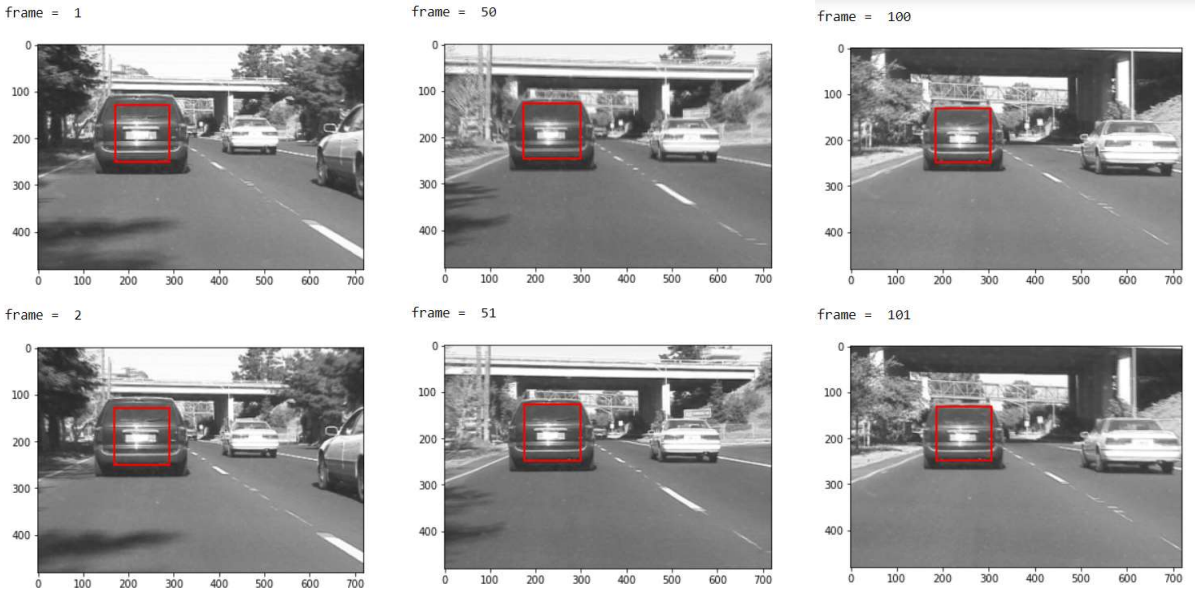
Q1.2

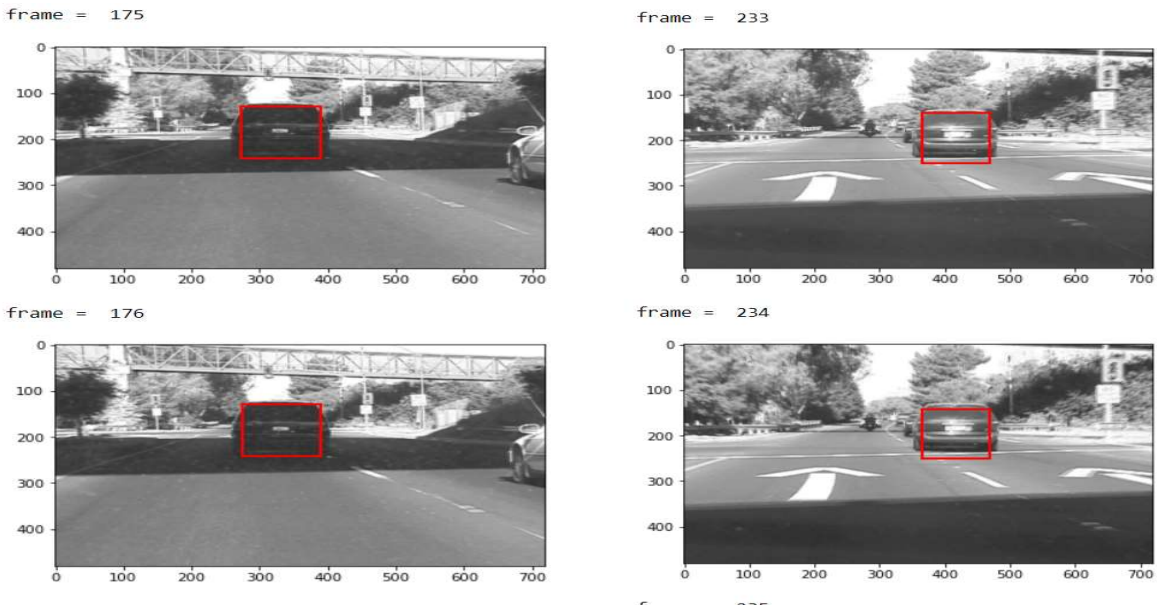
Code

In [28]:
```
def LucasKanadeAffine(It, It1, rect, thresh=0.025, maxIt=100):
    '''
    Q1.2: Lucas-Kanade Forward Additive Alignment with Affine MAtrix

      Inputs:
        It: template image
        It1: Current image
        rect: Current position of the object
        (top left, bottom right coordinates, x1, y1, x2, y2)
        thresh: Stop condition when dp is too small
        maxIt: Maximum number of iterations to run

      Outputs:
        M: Affine mtarix (2x3)
    '''

    # Set thresholds (you probably want to play around with the values)
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    threshold = thresh
    maxIters = maxIt
    i = 0
    x1, y1, x2, y2 = rect

    # ----- TODO -----
    # YOUR CODE HERE
    i = 0
    deltam = np.finfo('float').max * np.ones((2,3))

    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3]
    if (rect[0] > rect[2]):
        x1 = rect[2]
        x2 = rect[0]
    if (rect[1] > rect[3]):
        x1 = rect[3]
        x2 = rect[1]

    It_int = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]), It)
    It1_int = RectBivariateSpline(np.arange(It1.shape[0]), np.arange(It1.shape[1]), It1)
```

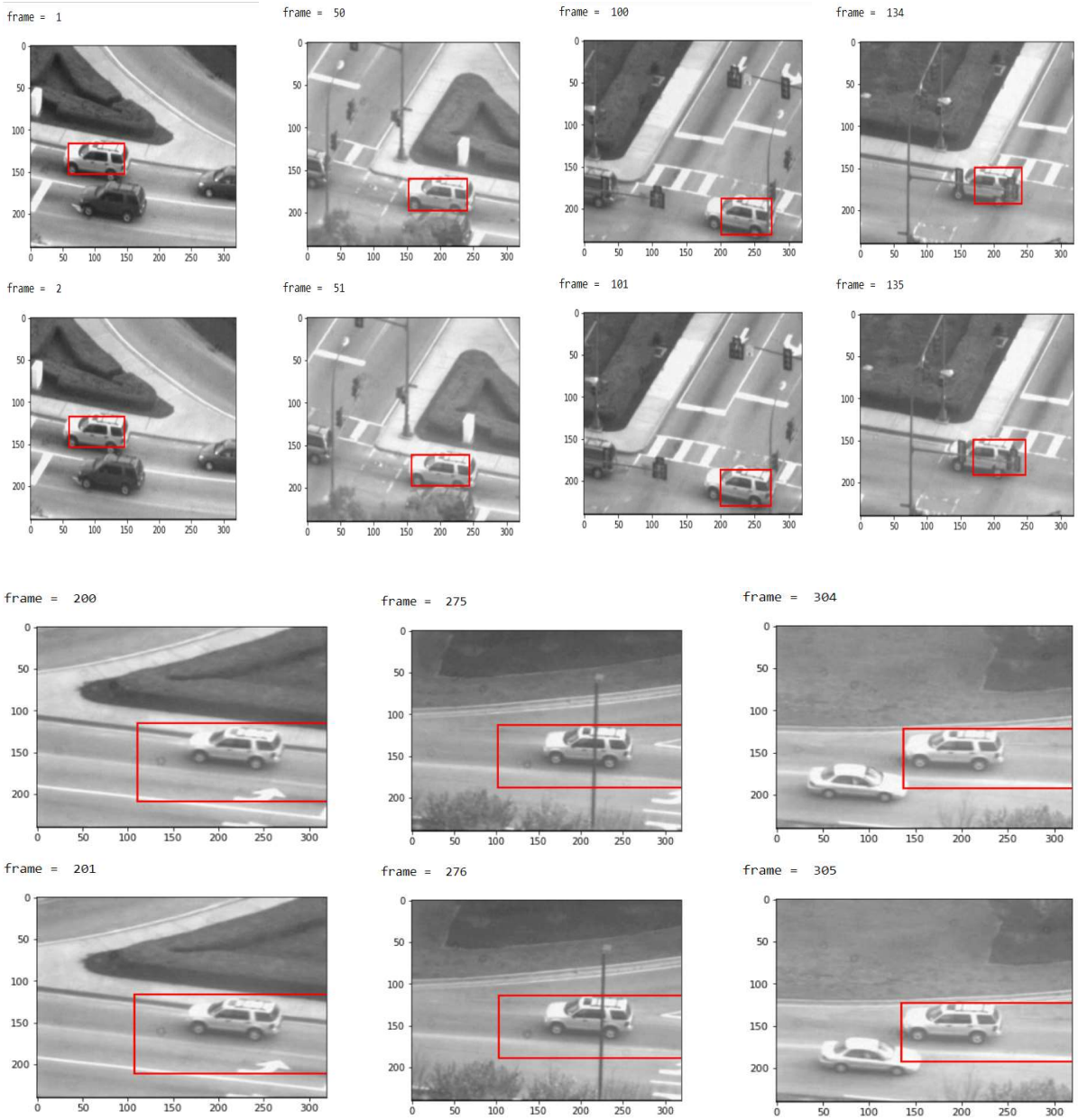
```
    xt = np.arange(x1, x2 + 0.5)
    yt = np.arange(y1 , y2 + 0.5)
    xg,yg = np.meshgrid(xt, yt)
    xg = xg.flatten()
    yg = yg.flatten()
    #print(xg.shape)
    #print(yg.shape)
    template = It_int.ev(yg, xg)
    #print(template.shape)
    h, w = It1.shape
    x = np.arange(w)
    y = np.arange(h)
    while (np.linalg.norm(deltam) > threshold and i < maxIt):
        #print(i+1)

        xgi = M[0,0] * xg + M[0,1] * yg + M[0,2]
        ygi = M[1,0] * xg + M[1,1] * yg + M[1,2]

        ind = (xgi > 0) & (ygi > 0 ) & (xgi < It1.shape[1]) & (ygi < It1.shape[0])
        #print(ind)
        #print(ygi.shape)
        xgi = xgi[ind]
        ygi = ygi[ind]
        #print(ygi.shape)
        patch = It1_int.ev(ygi, xgi)
        #print("size of patch", patch.shape)
        b = (template[ind] - patch).flatten()
        #print(b.shape)
        Ix = It1_int.ev(ygi, xgi, dx = 0, dy = 1)
        Iy = It1_int.ev(ygi, xgi, dx = 1, dy = 0)

        Iy = Iy.flatten()
        Ix = Ix.flatten()
        xgi = xgi.flatten()
        ygi = ygi.flatten()
        A = np.array([Ix*xgi, ygi*Ix, Ix, xgi*Iy, ygi*Iy, Iy]).T


        #print(gradI.shape)
        deltap = np.linalg.lstsq(A,b,rcond=None)[0]
        M = M + deltap.reshape((2,3))
        #print(deltap)
        #print(p)
        #print(M)
        i+=1


    return M
```

1) Car 1

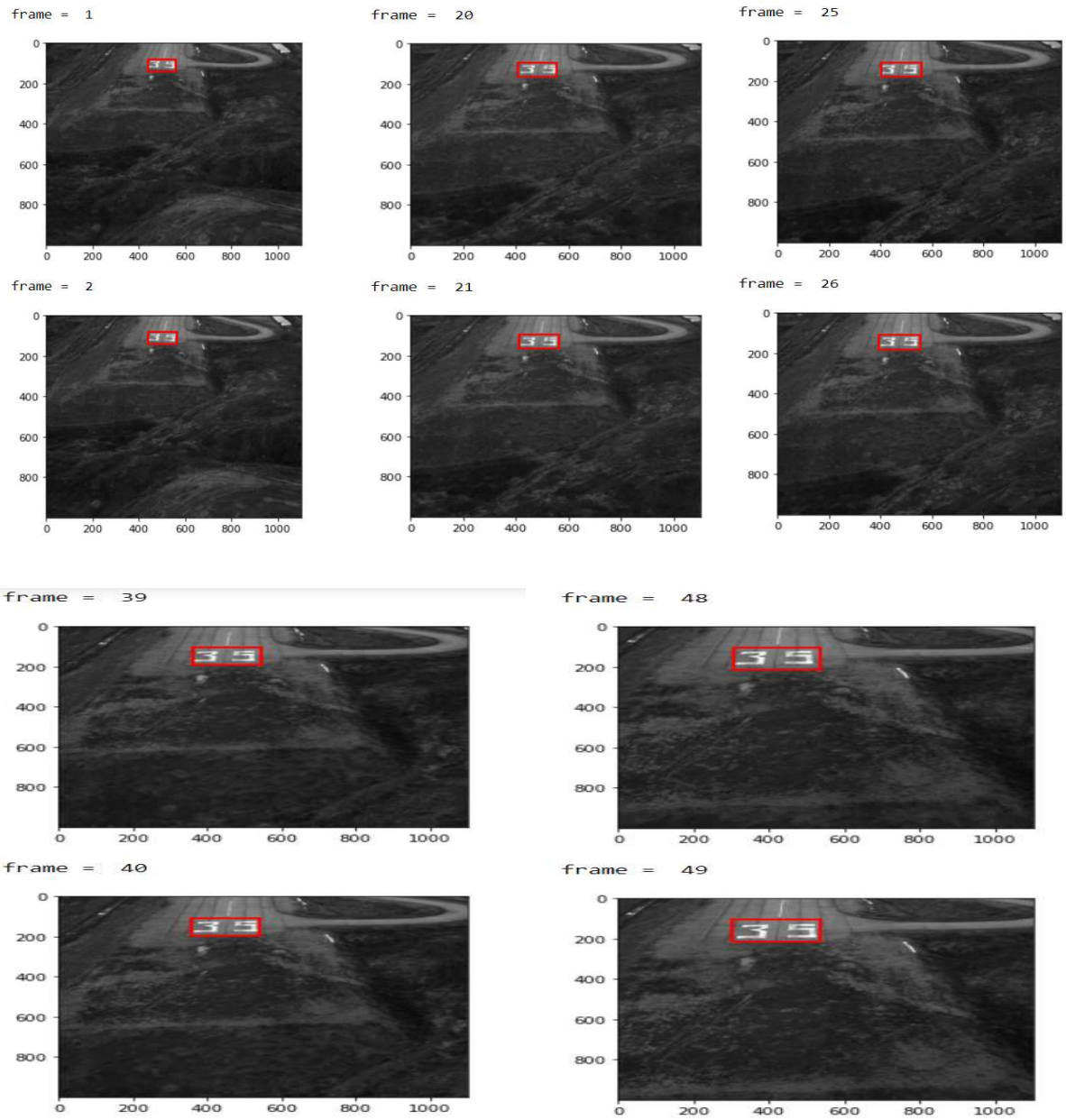

frame = 1
frame = 50
frame = 100
frame = 2
frame = 51
frame = 101


frame = 175
frame = 233
frame = 176
frame = 234

2) Car 2

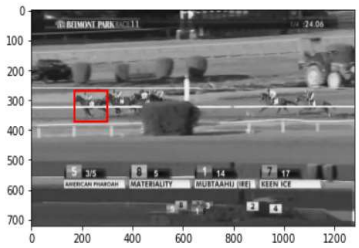

frame = 1
frame = 50
frame = 100
frame = 134
frame = 2
frame = 51
frame = 101
frame = 135
frame = 200
frame = 275
frame = 304
frame = 201
frame = 276
frame = 305

3) Landing

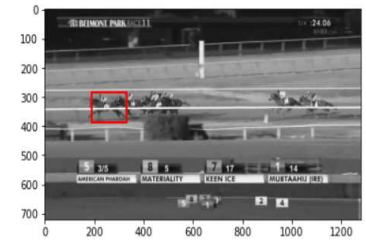

frame = 1
frame = 20
frame = 25
frame = 2
frame = 21
frame = 26
frame = 39
frame = 48
frame = 40
frame = 49

4) Racing

```
frame =  1
```

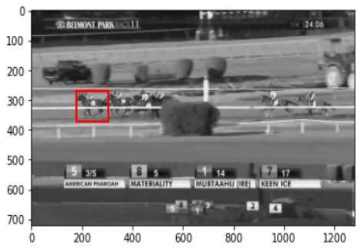


```
frame =  19
```



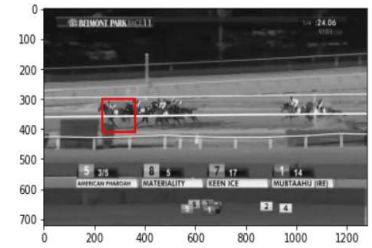
```
frame =  34
```

```
frame =  2
```

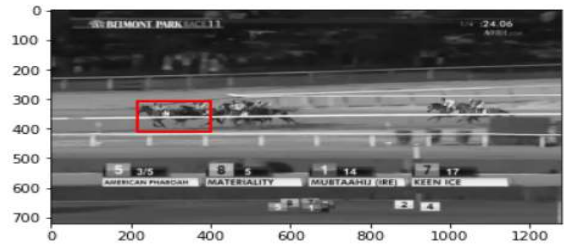


```
frame =  20
```

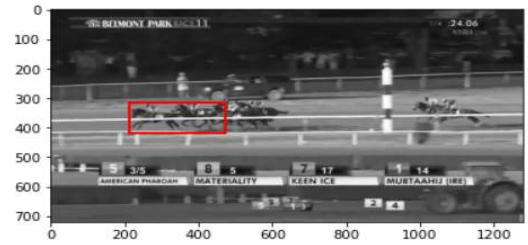
```
frame =  35
```



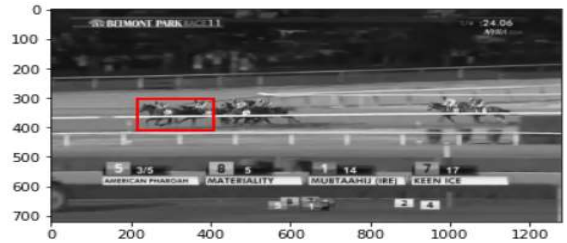
```
frame =  49
```



```
frame =  66
```



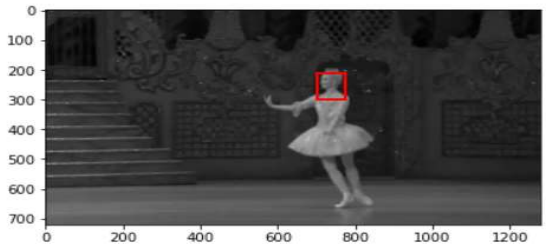
```
frame =  50
```



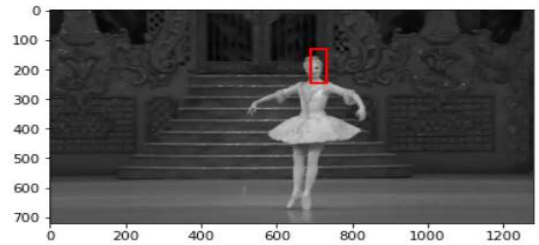
```
frame =  67
```

5) Ballet

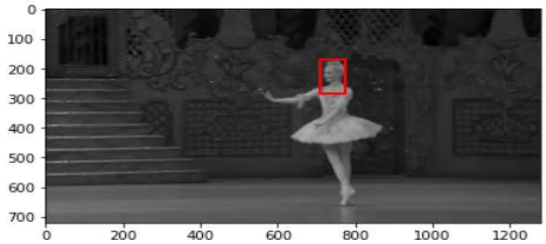
frame = 1



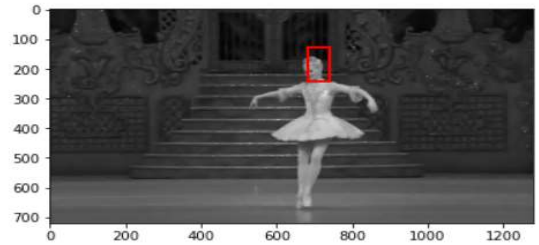
frame = 20



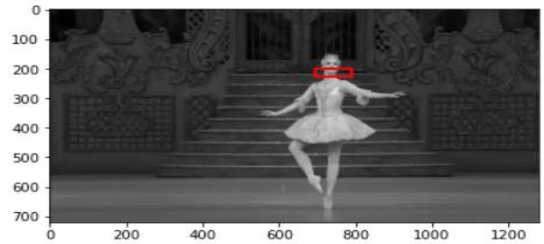
frame = 2



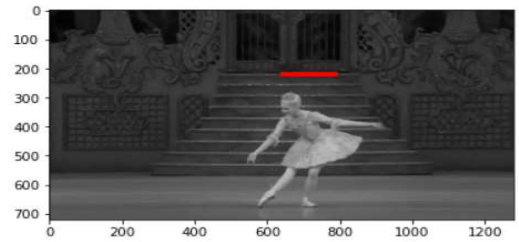
frame = 21



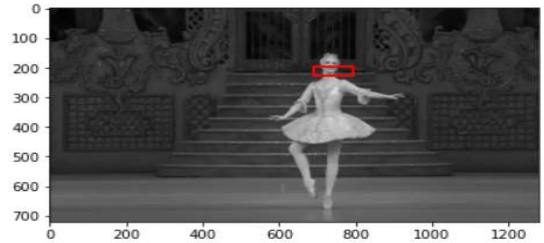
frame = 30



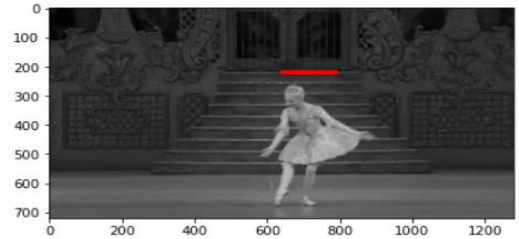
frame = 40



frame = 31



frame = 41



frame = 42

Q2.1

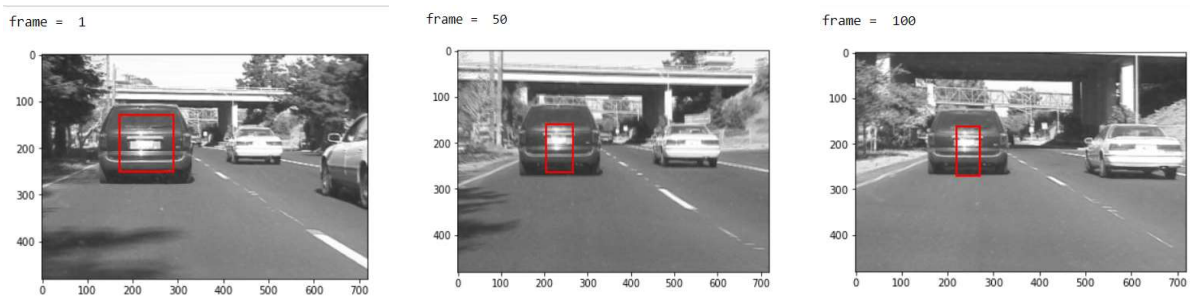
Code

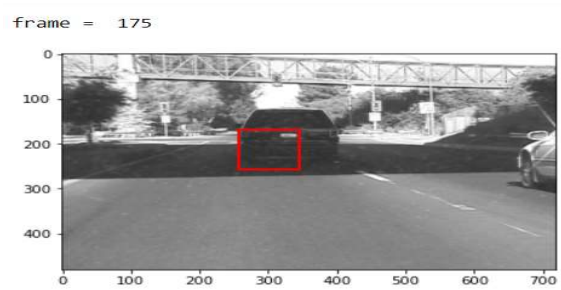
In [ ]:
```
def InverseCompositionAffine(It, It1, rect, thresh=.025, maxIt=15):
    '''
    Q2.1: Matthew-Bakers Inverse Compositional Alignment with Affine MAtrix

      Inputs:
        It: template image
        It1: Current image
        rect: Current position of the object
        (top left, bottom right coordinates, x1, y1, x2, y2)
        thresh: Stop condition when dp is too small
        maxIt: Maximum number of iterations to run

      Outputs:
        M: Affine mtarix (2x3)
    '''

    # Set thresholds (you probably want to play around with the values)
    #M = np.zeros((2,3))
#     threshold = thresh
#     maxIters = maxIt
#     i = 0
#     x1, y1, x2, y2 = rect

    # ----- TODO -----
    # YOUR CODE HERE
    #raise NotImplementedError()
    M = np.array([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
    threshold = thresh
    maxIters = maxIt
    i = 0
    x1, y1, x2, y2 = rect

    # ----- TODO -----
    # YOUR CODE HERE
    i = 0
    deltam = np.finfo('float').max * np.ones((2,3))
    x1, y1, x2, y2 = rect[0], rect[1], rect[2], rect[3]


    It_int = RectBivariateSpline(np.arange(It.shape[0]), np.arange(It.shape[1]), It)
    It1_int = RectBivariateSpline(np.arange(It1.shape[0]), np.arange(It1.shape[1]), It1)

    xt = np.arange(x1, x2 + 0.1)
    yt = np.arange(y1 , y2 + 0.1)

    xg,yg = np.meshgrid(xt, yt)
```

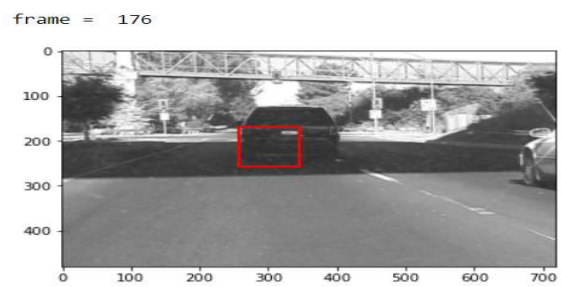
```
    xg = xg.flatten()
    yg = yg.flatten()

    # Taking the derivative of the template
    Ix = It_int.ev(yg, xg, dx = 0, dy = 1)
    Iy = It_int.ev(yg, xg, dx = 1, dy = 0)
    Iy = Iy.flatten()
    Ix = Ix.flatten()


    #Evaluating the Jacobian and hessian
    A = np.array([Ix*xg, yg*Ix, Ix, xg*Iy, yg*Iy, Iy]).T #  6 x N
    #print(A.shape)
    H = A.T @ A # 6 x 6
    #print(H.shape)

    #print(xg.shape)
    #print(yg.shape)
    template = It_int.ev(yg, xg)

    h, w = It1.shape
    x = np.arange(w)
    y = np.arange(h)

    while (np.linalg.norm(deltam) > threshold and i < maxIt):
        #print(i+1)
        #print("error", np.linalg.norm(deltam))
        xgi = M[0,0] * xg + M[0,1] * yg + M[0,2]
        ygi = M[1,0] * xg + M[1,1] * yg + M[1,2]
        xgi = xgi.flatten()
        ygi = ygi.flatten()
        ind = (xgi > 0) & (ygi > 0 ) & (xgi < It1.shape[1]) & (ygi < It1.shape[0])
        #print(ind)
        #print(ygi.shape)
        xgi = xgi[ind]
        ygi = ygi[ind]

        patch = It1_int.ev(ygi, xgi)

        err = ((- template[ind] + patch)).reshape(-1,1)
        #print(err.shape)
        #print(A.shape)

        deltam =   A[ind, :].T @ err  # 6x6 6xN N x 1 = 6 x 1
        #print(err.shape)
        #print(deltam.shape)
        #print(np.linalg.inv(H).shape)
        deltam =  np.linalg.pinv(H) @ deltam
```

```
        M = np.vstack((np.reshape(M, (2, 3)), np.array([[0, 0, 1]])))
        deltam = np.vstack((np.reshape(deltam, (2, 3)), np.array([[0, 0, 1]])))
        deltam[0, 0] += 1
        deltam[1, 1] += 1
        M = np.dot(M, np.linalg.pinv(deltam))
        M = M[:2, :]
        #print(gradI.shape)

        #print(deltap)
        #print(p)
        #print(M)
        i+=1


    return M
```

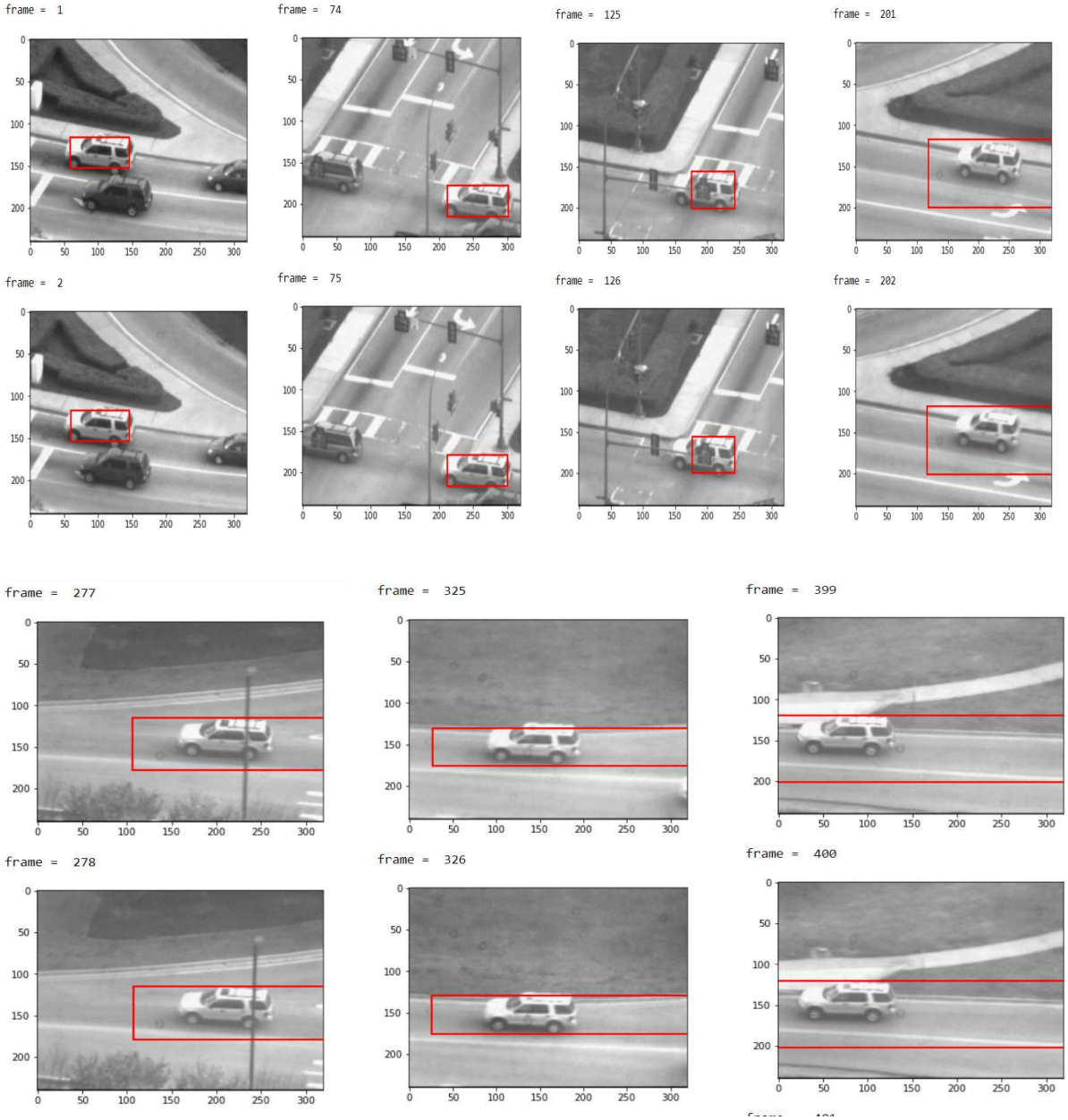
1) Car 1



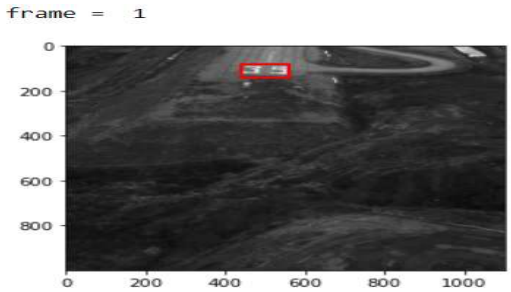


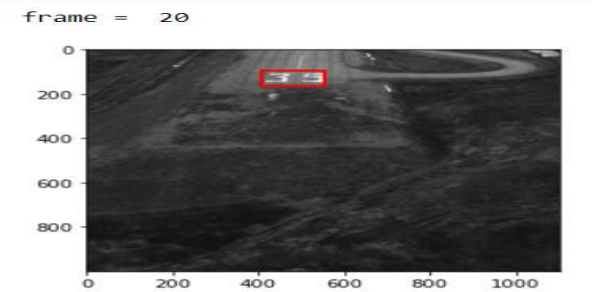


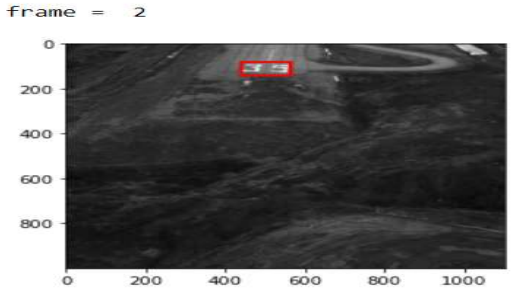
2) Car 2

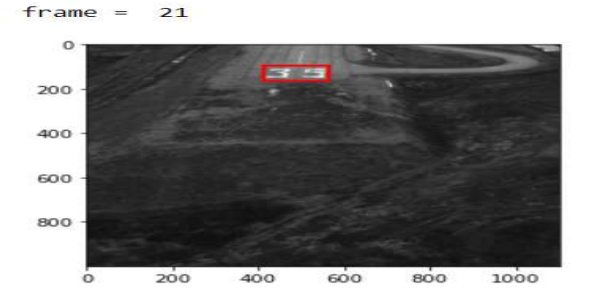

frame = 1
frame = 74
frame = 125
frame = 201
frame = 2
frame = 75
frame = 126
frame = 202
frame = 277
frame = 325
frame = 399
frame = 278
frame = 326
frame = 400

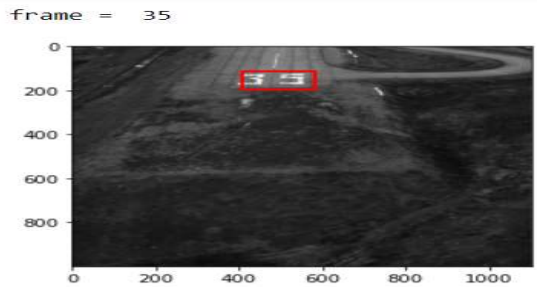
3) Landing

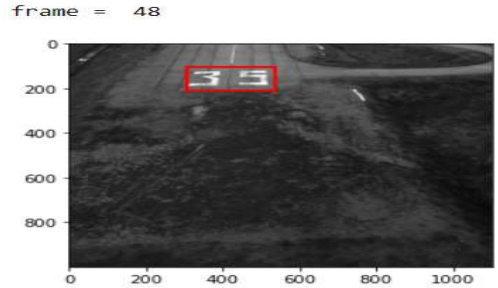

frame = 1

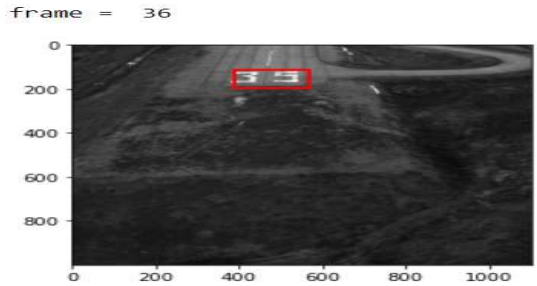

frame = 20

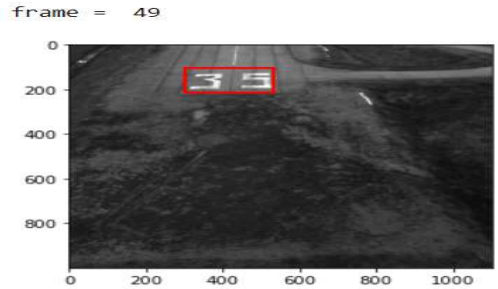

frame = 2


frame = 21


frame = 35


frame = 48

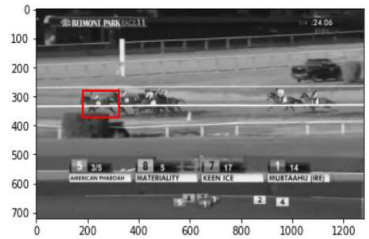

frame = 36


frame = 49

4) Racing

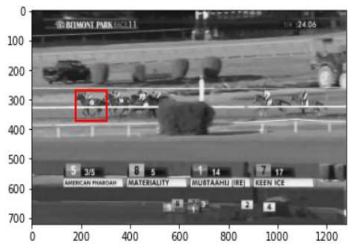
```
frame =  1
```

```
frame =  15
```



```
frame =  24
```

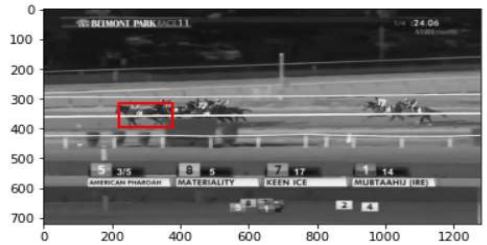
```
frame =  2
```



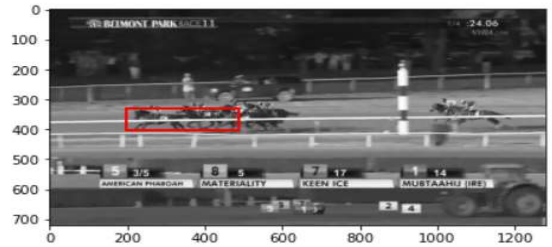
```
frame =  16
```

```
frame =  25
```

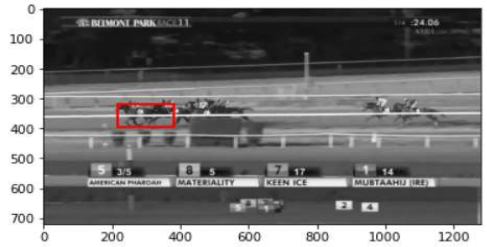
```
frame =  39
```



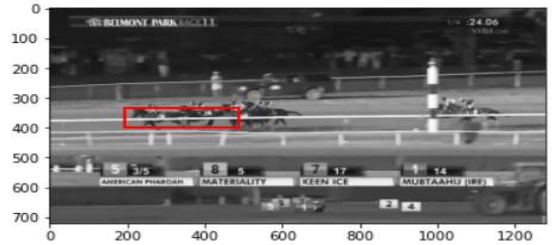
```
frame =  66
```



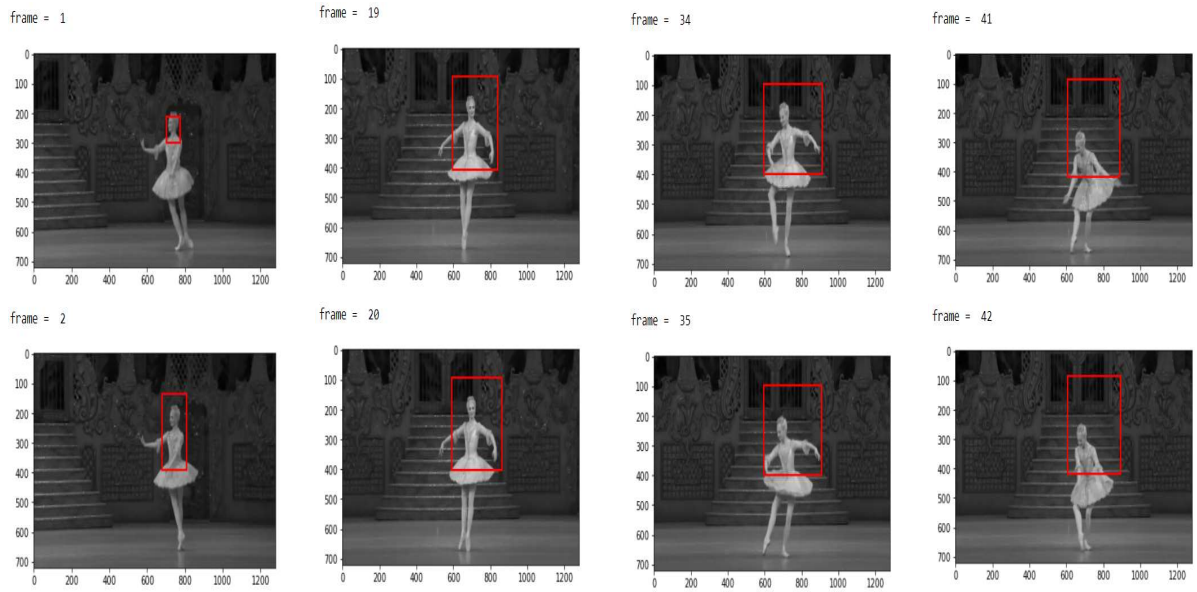
```
frame =  40
```



```
frame =  67
```



5) Ballet



Q2.2

All the three algorithm work best with landing and race video as the movement of the frame is consistent and gradual, hence assuption of the lucas kannade is valid and the brightness consistency equation is not failing.

For Lucas Kannade Translation: as the there isnt much transormation parameters (2 parameters), the bounding boxes remain rectangular and error for calculating those parameters is lesser. Hence the videos with no change in size or other transformations can be efficently tracked .

For Lucas Kanade Affine, tracking for ballet and car 2 are poor as the movement of ballet and car is much higher in last few frames and this violates the assumptions of lucas kannade there by failing the brightness consistency equation.

Lucas kanade in overall is very slow because of more calculation parameters in the iteration compare to mathew bakers.

In mathew bakers, the results are similar to lucas kanade affine but a bit worse as the calculation of H inverse and double warping make it less accurate but its faster than lucas kanade.

Q3

YOUR ANSWER HERE