CMU Computer Vision HW6

Jiaqi Geng

Dec 6th 2020

1 Theory Questions

Q1.1

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

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

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

Q1.2

The complexity of matrix multiplication between a $n \times m$ matrix and a $m \times p$ matrix is $O(nmp)$.

For each iteration,
We first need to warp the pixels in template image.
There are n number of pixels in template image,
and there are p number of parameters for the warp matrix.
Thus, the complexity of this step is $O(np)$

Then, we calculate the error image.
Since there are n number of pixels both in the template image and in the warp image,
and we do subtraction for each pixel,
the complexity for this step is $O(n)$.

Then, we need to compute the gradient of the warped coordinates.
For each pixel, we will do constant amount of work to calculate the gradient, which is $O(n)$
We already know warping takes $O(np)$
Thus, the complexity of this step is $O(np)$

Then, we need to evaluate jacobian for each pixel in the template.
For one pixel, the complexity is $O(p)$
Thus, the complexity of this step is $O(np)$

Next, we are gonna calculate $\nabla I \frac{\partial W}{\partial p}$
Since there are n number of gradient in ∇I,
and there are p number of parameters for each pixel,
the complexity of this step is $O(np)$.

Then, we need to compute the hessian.
For each pixel, $\nabla I \frac{\partial W}{\partial p}$ has a shape of $(1, p)$
Therefore, to get $[\nabla I \frac{\partial W}{\partial p}]^T [\nabla I \frac{\partial W}{\partial p}]$, the time complexity is p^2
Since we have n number of pixels,
the complexity of this step is $O(np^2)$.

Next, we need to compute the inverse of Hessian.
Hessian matrix is a p by p matrix,
so the inverse of this matrix takes $O(p^3)$ time.

Now, we need to compute $\sum_x [\nabla I \frac{\partial W}{\partial p}]^T [T(x) - I(W(x;p))]$
For each pixel,
$\nabla I \frac{\partial W}{\partial p}$ is a 1 by p matrix, and $T(x) - I(W(x;p))$ is a scalar.
Thus, the complexity of this step is $O(np)$.

Now we can matrix multiply inverse of Hessian and $\sum_x [\nabla I \frac{\partial W}{\partial p}]^T [T(x) - I(W(x;p))]$
The inverse of H is a p by p matrix and and $\sum_x [\nabla I \frac{\partial W}{\partial p}]^T [T(x) - I(W(x;p))]$ is a p by 1 matrix

so the complexity of this step takes $O(p^2)$

Finally, we need to update p with Δp.
Since we have p number of parameters,
the complexity of this step is $O(p)$

Therefore, the total complexity for each iteration is $O(np^2 + p^3)$

Q1.3

For pre-computation,
First, we need to get the gradient of the template image,
since there are n number of pixels in the template,
and it takes constant time to get gradient for one pixel,
the complexity is $O(n)$.

Next, we evaluate the jacobian,
we have calculate the time complexity of this step in $Q1.2$,
the complexity is $O(np)$.

Next, we are gonna calculate $\nabla T \frac{\partial W}{\partial p}$
Since there are n number of gradient in ∇T,
and there are p number of parameters for each pixel,
the complexity of this step is $O(np)$.

Now we compute the hessian matrix.
Just like what we did in $Q1.2$,
this step takes $O(np^2)$.

Therefore, the pre-computation step takes $O(np^2)$

Now for each iteration,
We first need to warp the pixels in template image.
There are n number of pixels in template image,
and there are p number of parameters for the warp matrix.
Thus, the complexity of this step is $O(np)$

Then, we calculate the error image.
Since there are n number of pixels both in the template image and in the warp image,
and we do subtraction for each pixel,
the complexity for this step is $O(n)$.

Now, we need to compute $\sum_x [\nabla T \frac{\partial W}{\partial p}]^T [I(W(x;p)) - T(x)]$
For each pixel,
$\nabla T \frac{\partial W}{\partial p}$ is a 1 by p matrix, and $I(W(x;p)) - T(x)$ is a scalar.
Thus, the complexity of this step is $O(np)$.

Now we can matrix multiply inverse of Hessian and $\sum_x [\nabla I \frac{\partial W}{\partial p}]^T [T(x) - I(W(x;p))]$
It takes $O(p^3)$ time to compute the inverse of Hessian.
The inverse of H is a p by p matrix and and $\sum_x [\nabla I \frac{\partial W}{\partial p}]^T [T(x) - I(W(x;p))]$ is a p by 1 matrix
so the complexity of multiplication takes $O(p^2)$
and the entire step takes $O(p^3)$

Finally, we update the warp.

Since there are p parameters,
the multiplication between two matrix with p parameters takes $O(p^2)$ times

Thus, the total complexity is $O(np + p^3)$

We can see that
the run time of Matthews-Baker will be smaller (faster) than the run time of Lucas-Kanade method.

2 Implementing the Trackers

Q2.1

```
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
    h, w = It.shape

    rbs = RectBivariateSpline(np.arange(h), np.arange(w), It)
    rbs1 = RectBivariateSpline(np.arange(h), np.arange(w), It1)

    rect_xs = np.linspace(x1, x2, int(x2-x1))
    rect_ys = np.linspace(y1, y2, int(y2-y1))
    rect_xx, rect_yy = np.meshgrid(rect_xs, rect_ys)
    template = rbs.ev(rect_yy, rect_xx)

    delta_p = np.array([[10000], [10000]])
    counter = 0

    while np.sqrt(np.sum(delta_p ** 2)) >= threshold and \
            counter <= maxIters:

        rect_xx_, rect_yy_ = rect_xx + p[0], rect_yy + p[1]
        warped = rbs1.ev(rect_yy_, rect_xx_)
        error = template - warped

        Ix, Iy = rbs1.ev(rect_yy_, rect_xx_, 0, 1), \
                 rbs1.ev(rect_yy_, rect_xx_, 1, 0)
        Ix, Iy = Ix.reshape((-1, 1)), Iy.reshape((-1, 1))
        I = np.hstack((Ix, Iy))
```

```
        jacobian = np.eye(2)
        A = np.dot(I, jacobian)
        H = np.dot(A.T, A)

        delta_p = np.dot(np.dot(np.linalg.inv(H), A.T),
                         error.reshape((-1, 1)))
        p[0] += delta_p[0, 0]
        p[1] += delta_p[1, 0]
        counter += 1

    return p
```

Q2.2

```
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
    h, w = It.shape

    # put your implementation here
    rbs = RectBivariateSpline(np.arange(h), np.arange(w), It)
    rbs1 = RectBivariateSpline(np.arange(h), np.arange(w), It1)

    rect_xs = np.linspace(x1, x2, int(x2-x1))
    rect_ys = np.linspace(y1, y2, int(y2-y1))
    rect_xx, rect_yy = np.meshgrid(rect_xs, rect_ys)

    template = rbs.ev(rect_yy, rect_xx)

    delta_p = np.array([[10000], [10000], [10000], [10000], [10000], [10000]])
    counter = 0

    while np.sqrt(np.sum(delta_p ** 2)) >= threshold and counter <= maxIters:
        rect_xx_ = (1 + p[0, 0]) * rect_xx + p[1, 0] * rect_yy + p[2, 0]
        rect_yy_ = p[3, 0] * rect_xx + (1 + p[4, 0]) * rect_yy + p[5, 0]

        warped = rbs1.ev(rect_yy_, rect_xx_)
        error = template - warped

        x_flatten, y_flatten = rect_xx_.reshape((-1, 1)), \
                               rect_yy_.reshape((-1, 1))

        jacobian = np.zeros((x_flatten.shape[0] * 2, 6))
```

```
    jacobian[::2] = np.hstack((x_flatten,
                               np.zeros((x_flatten.shape[0], 1)),
                               y_flatten,
                               np.zeros((x_flatten.shape[0], 1)),
                               np.ones((x_flatten.shape[0], 1)),
                               np.zeros((x_flatten.shape[0], 1))))

    jacobian[1::2] = np.hstack((np.zeros((x_flatten.shape[0], 1)),
                                x_flatten,
                                np.zeros((x_flatten.shape[0], 1)),
                                y_flatten,
                                np.zeros((x_flatten.shape[0], 1)),
                                np.ones((x_flatten.shape[0], 1))))

    Ix, Iy = rbs1.ev(rect_yy_, rect_xx_, 0, 1), \
             rbs1.ev(rect_yy_, rect_xx_, 1, 0)
    Ix, Iy = Ix.reshape((-1, 1)), Iy.reshape((-1, 1))
    I = np.hstack((Ix, Iy))

    A = np.zeros((x_flatten.shape[0], 6))
    for i in range(x_flatten.shape[0]):
        A[i, :] = np.dot(I[i, :],
                         jacobian[2*i: 2*i+2, :]).reshape((1, -1))

    H = np.dot(A.T, A)
    delta_p = np.dot(np.dot(np.linalg.inv(H), A.T),
                     error.reshape((-1, 1)))

    p[0] += delta_p[0, 0]
    p[1] += delta_p[2, 0]
    p[2] += delta_p[4, 0]
    p[3] += delta_p[1, 0]
    p[4] += delta_p[3, 0]
    p[5] += delta_p[5, 0]
    counter += 1

M = np.array([[1.0 + p[0], p[1], p[2]],
              [p[3], 1.0 + p[4], p[5]]]).reshape(2, 3)

return M
```

Q2.3

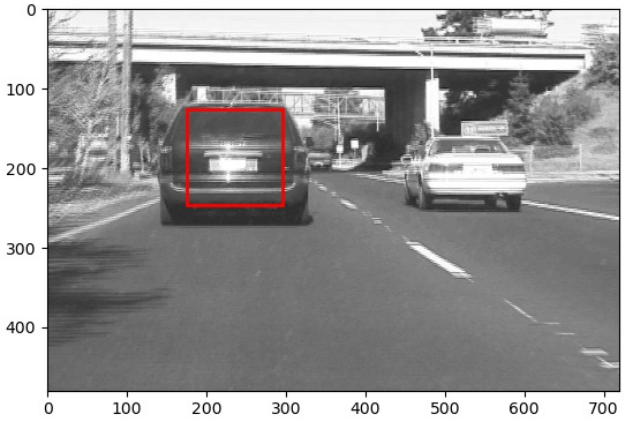
```
import numpy as np
from scipy.interpolate import RectBivariateSpline


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
    h, w = It.shape

    # put your implementation here
    rbs = RectBivariateSpline(np.arange(h), np.arange(w), It)
    rbs1 = RectBivariateSpline(np.arange(h), np.arange(w), It1)

    rect_xs = np.linspace(x1, x2, int(x2-x1))
    rect_ys = np.linspace(y1, y2, int(y2-y1))
    rect_xx, rect_yy = np.meshgrid(rect_xs, rect_ys)
    template = rbs.ev(rect_yy, rect_xx)

    Tx, Ty = rbs.ev(rect_yy, rect_xx, 0, 1), rbs.ev(rect_yy, rect_xx, 1, 0)
    Tx, Ty = Tx.reshape((-1, 1)), Ty.reshape((-1, 1))
    T_grad = np.hstack((Tx, Ty))

    x_flatten, y_flatten = rect_xx.reshape((-1, 1)), rect_yy.reshape((-1, 1))
    jacobian = np.zeros((x_flatten.shape[0] * 2, 6))
    jacobian[::2] = np.hstack((x_flatten, np.zeros((x_flatten.shape[0], 1)),
                               y_flatten, np.zeros((x_flatten.shape[0], 1)),
                               np.ones((x_flatten.shape[0], 1)),
                               np.zeros((x_flatten.shape[0], 1))))

    jacobian[1::2] = np.hstack((np.zeros((x_flatten.shape[0], 1)), x_flatten,
                                np.zeros((x_flatten.shape[0], 1)), y_flatten,
                                np.zeros((x_flatten.shape[0], 1)),
                                np.ones((x_flatten.shape[0], 1))))
```

```
J = np.zeros((x_flatten.shape[0], 6))
for i in range(x_flatten.shape[0]):
    J[i, :] = np.dot(T_grad[i, :],
                     jacobian[2 * i: 2 * i + 2, :]).reshape((1, -1))

H = np.dot(J.T, J)

counter = 0
delta_p = np.array([[10000], [10000], [10000], [10000], [10000], [10000]])

W = np.eye(3)

while np.sqrt(np.sum(delta_p ** 2)) >= threshold and counter <= maxIters:
    rect_xx_ = W[0, 0] * rect_xx + W[0, 1] * rect_yy + W[0, 2]
    rect_yy_ = W[1, 0] * rect_xx + W[1, 1] * rect_yy + W[1, 2]

    warped = rbs1.ev(rect_yy_, rect_xx_)

    error = warped - template
    delta_p = np.dot(np.dot(np.linalg.inv(H), J.T),
                     error.reshape((-1, 1)))

    original_W = W
    W = np.array([[1.0 + delta_p[0, 0], delta_p[2, 0], delta_p[4, 0]],
                  [delta_p[1, 0], 1.0 + delta_p[3, 0], delta_p[5, 0]],
                  [0.0, 0.0, 1.0]]).reshape(3, 3)

    W = np.dot(original_W, np.linalg.inv(W))
    counter += 1

return W[:2, :]
```

Q2.4

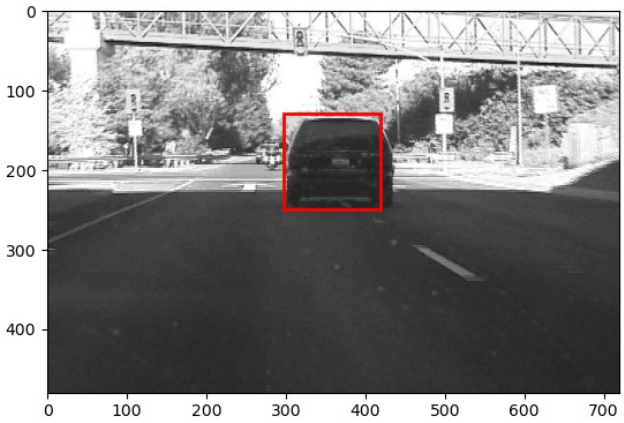
Lucas-Kanade Forward Additive Alignment with Translation:

Car1:

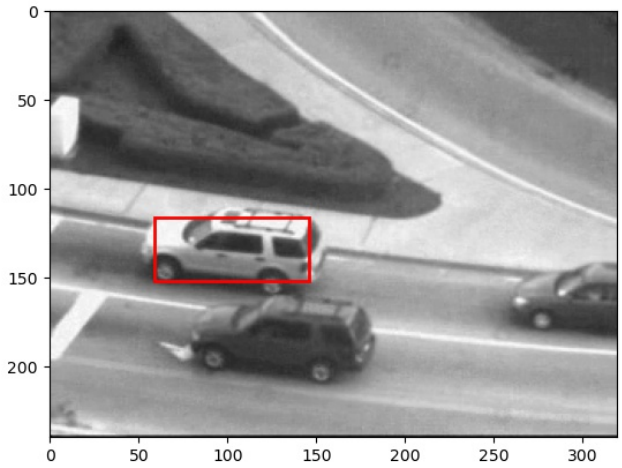


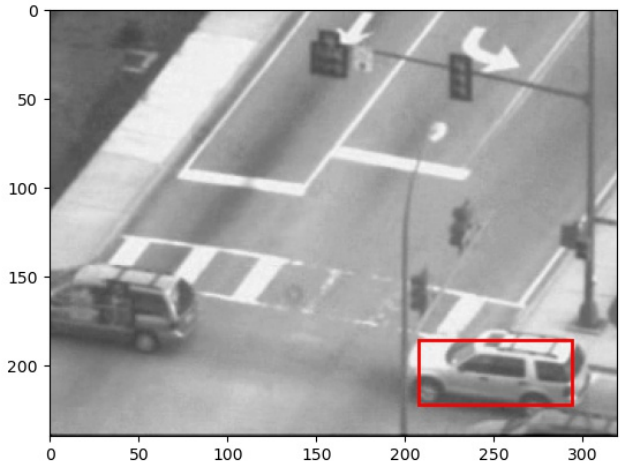

0
100
200
300
400
0
100
200
300
400
500
600
700

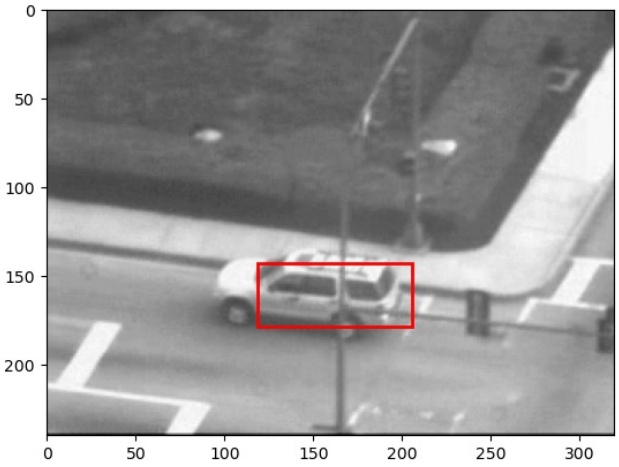

0
100
200
300
400
0
100
200
300
400
500
600
700

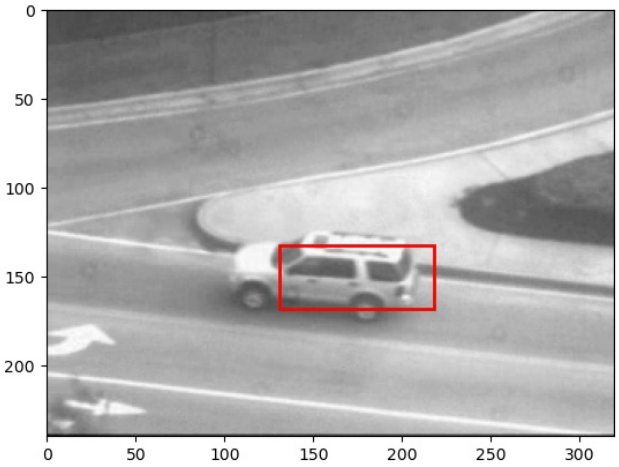

0
100
200
300
400
0
100
200
300
400
500
600
700

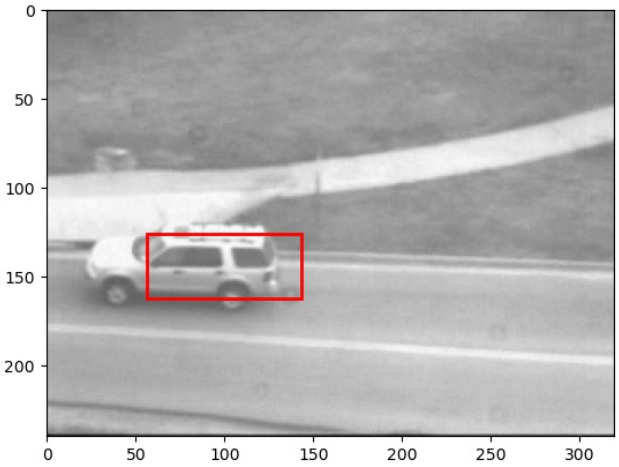
Car2:


0
50
100
150
200
0
50
100
150
200
250
300

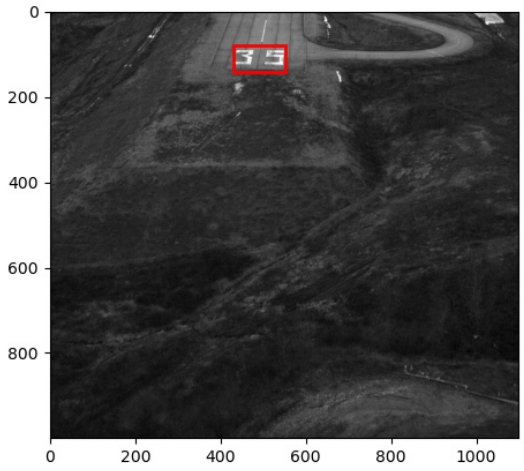

0
50
100
150
200
0
50
100
150
200
250
300

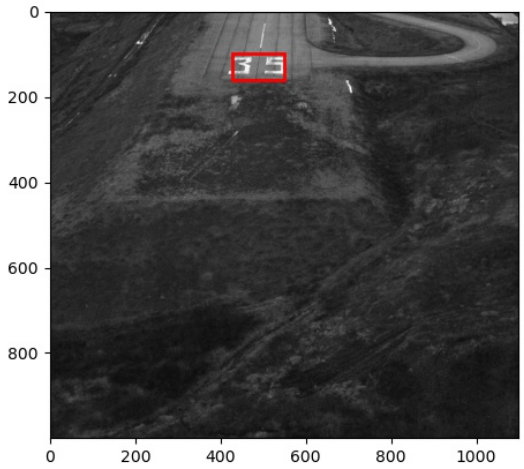

0
50
100
150
200
0
50
100
150
200
250
300

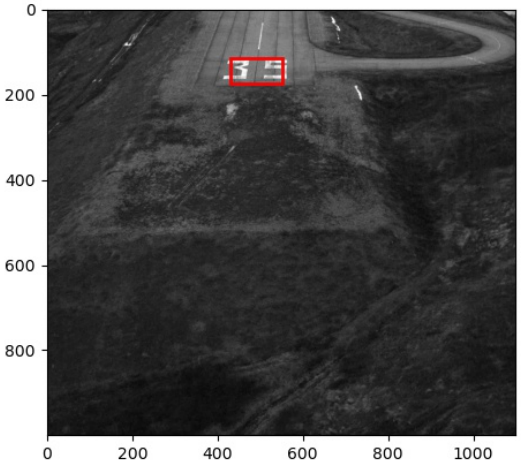

0
50
100
150
200
0
50
100
150
200
250
300

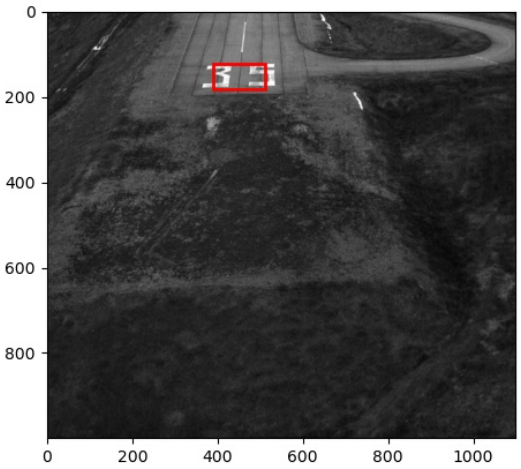

0
50
100
150
200
0
50
100
150
200
250
300

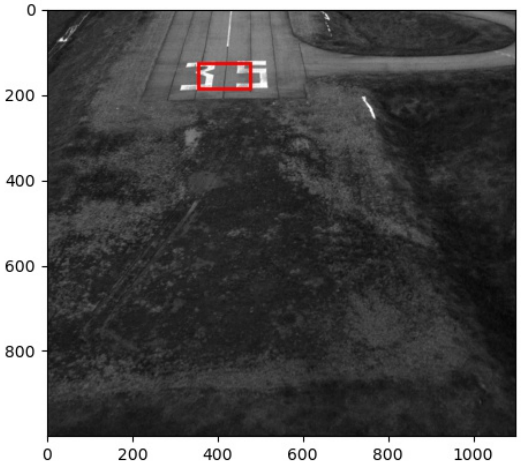
Landing:


0
200
400
600
800
0
200
400
600
800
1000

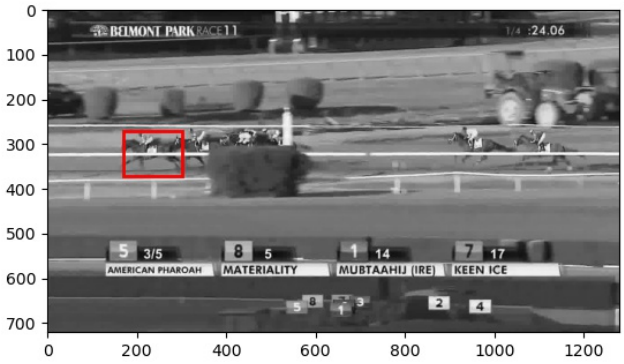

0
200
400
600
800
0
200
400
600
800
1000

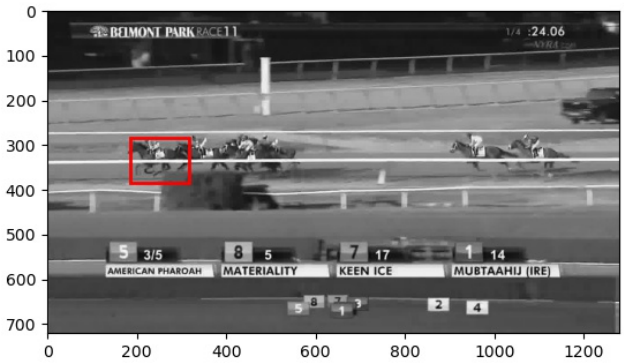

0
200
400
600
800
0
200
400
600
800
1000

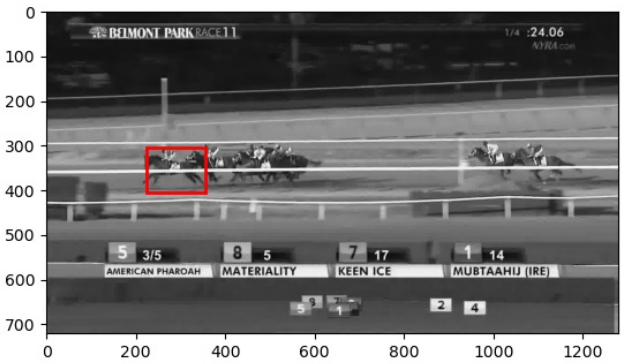

0
200
400
600
800
0
200
400
600
800
1000

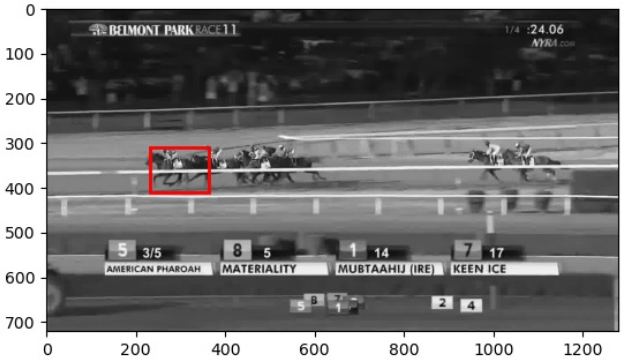

0
200
400
600
800
0
200
400
600
800
1000

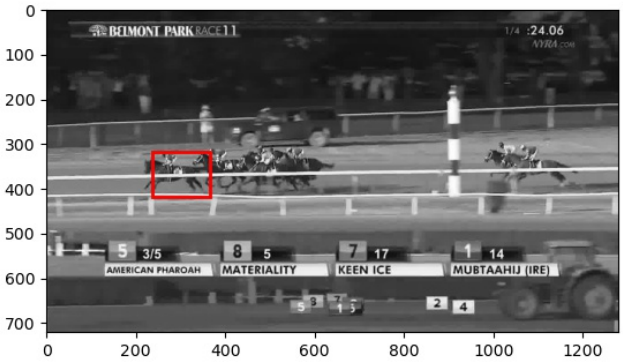
Race:


BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

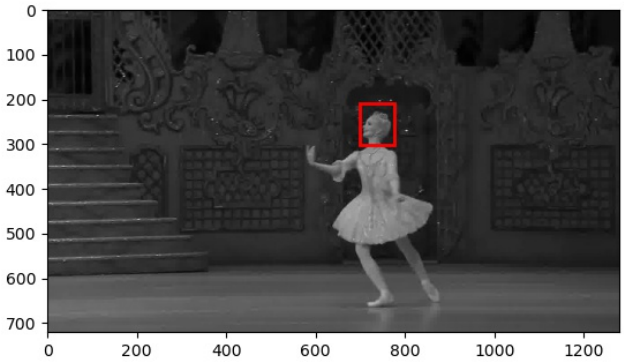

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

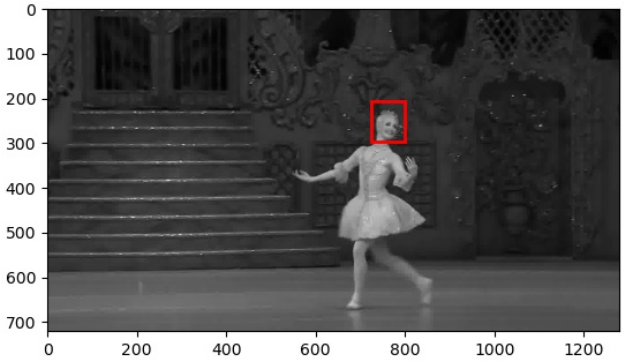

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

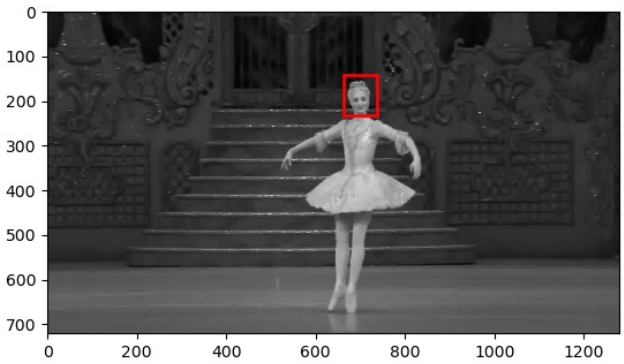

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE

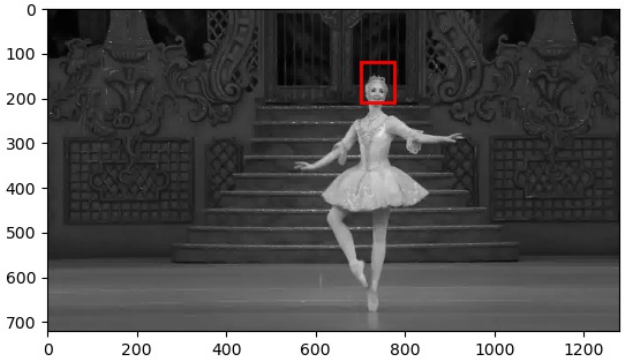

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

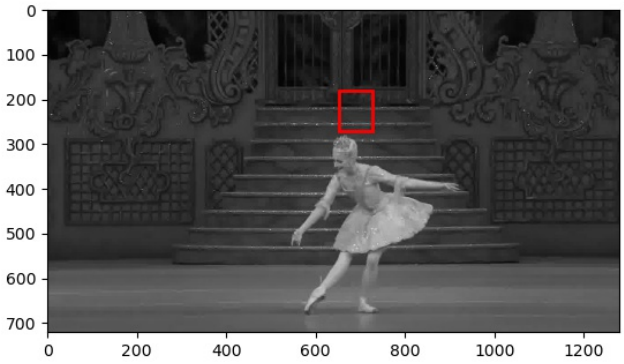
ballet:


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

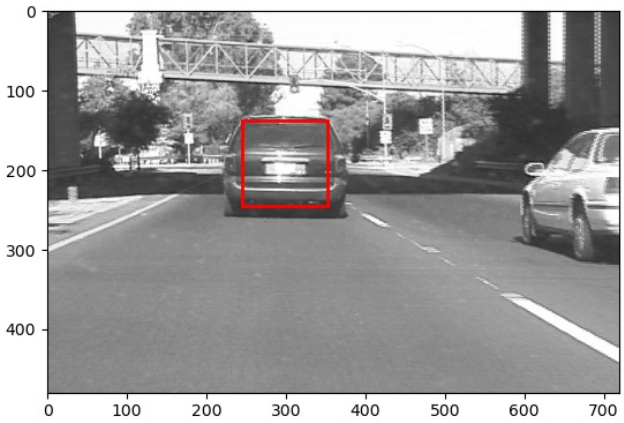

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

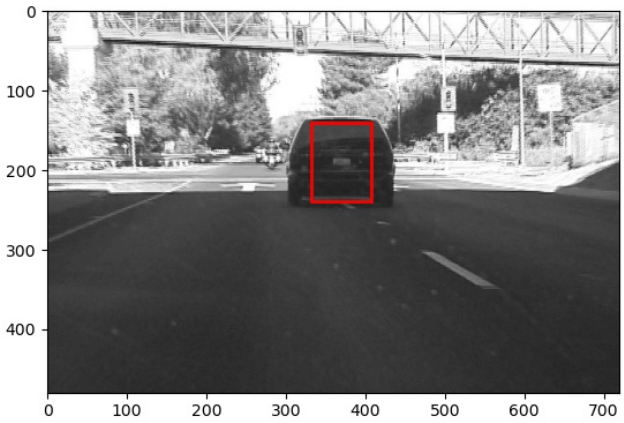

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Lucas-Kanade Forward Additive Alignment with Affine Transformation

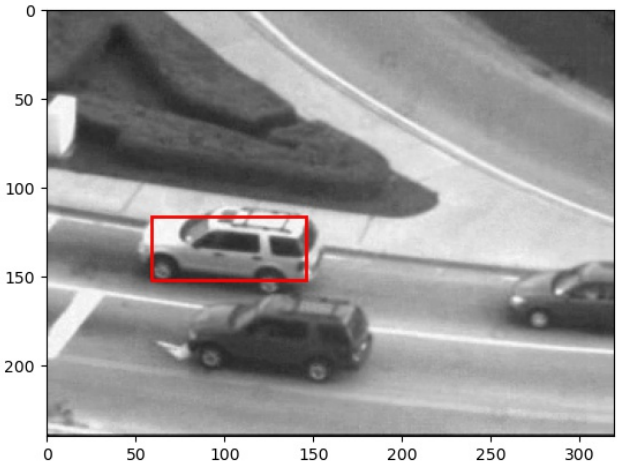
Car1:

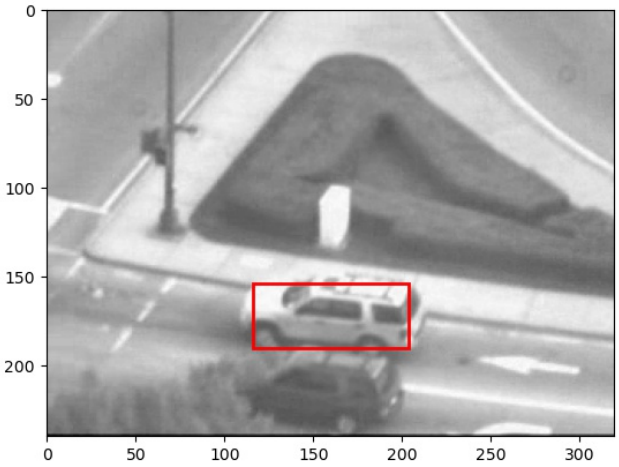

0
100
200
300
400
0
100
200
300
400
500
600
700


0
100
200
300
400
0
100
200
300
400
500
600
700

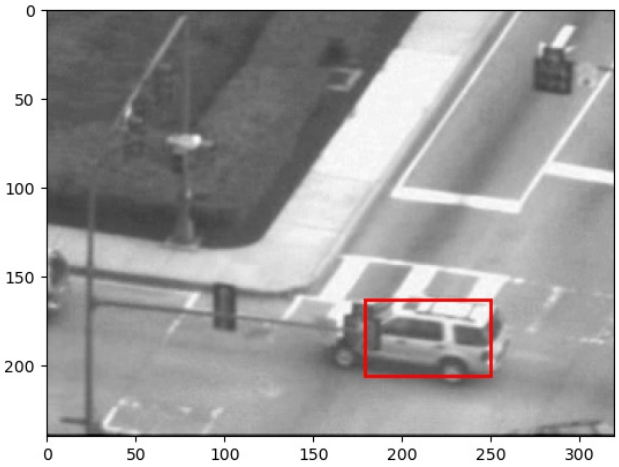

0
100
200
300
400
0
100
200
300
400
500
600
700

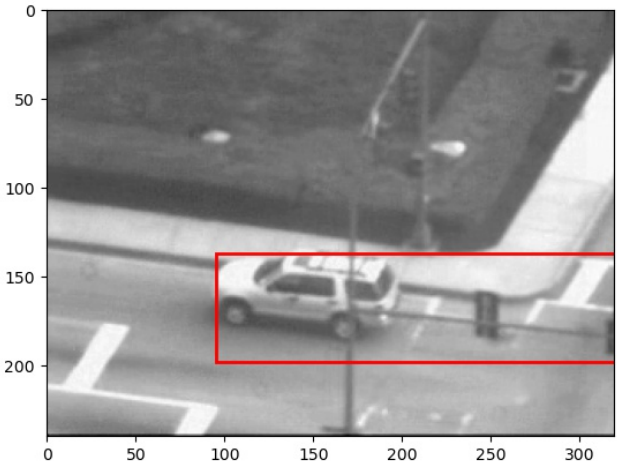
Car2 (only to frame 161, results not ideal afterwards):



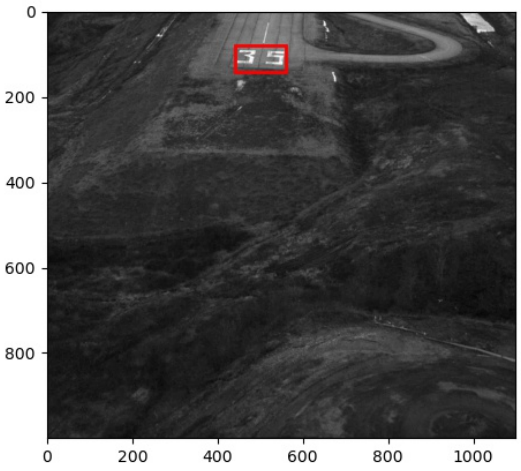


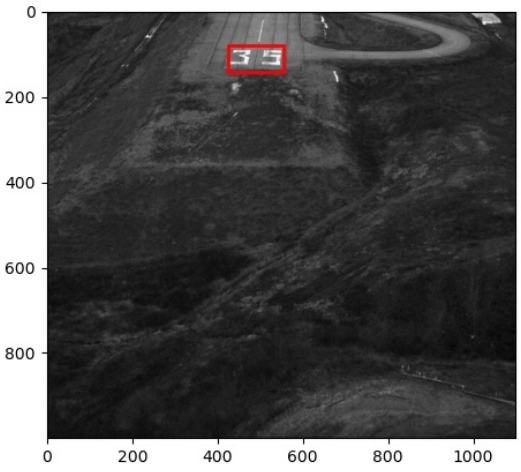

0
50
100
150
200
0
50
100
150
200
250
300

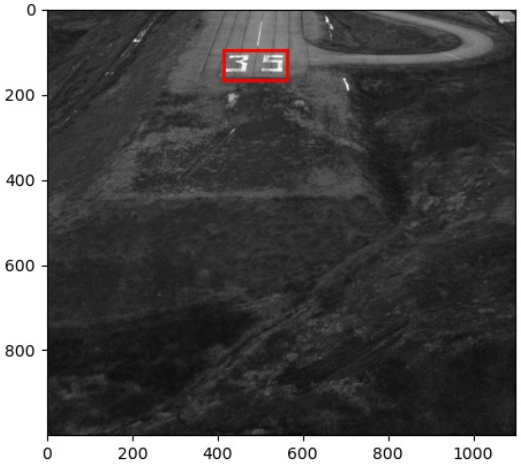

0
50
100
150
200
0
50
100
150
200
250
300

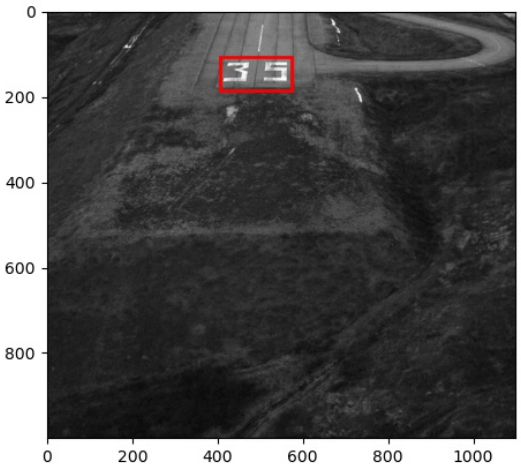

0
50
100
150
200
0
50
100
150
200
250
300

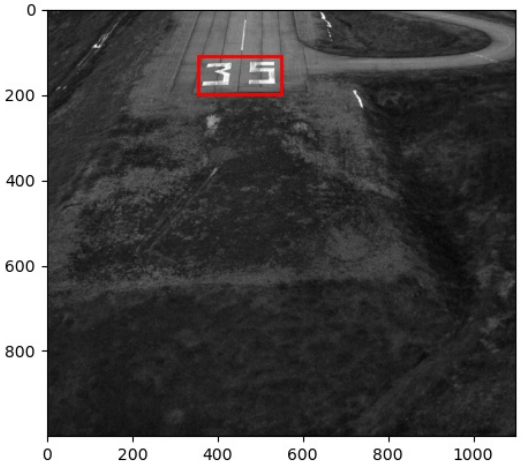
Landing:


0
200
400
600
800
0
200
400
600
800
1000

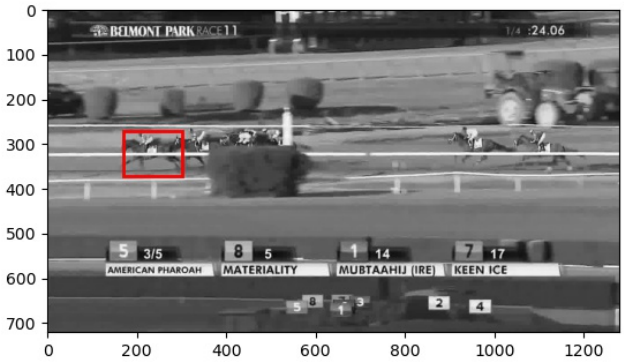

0
200
400
600
800
0
200
400
600
800
1000

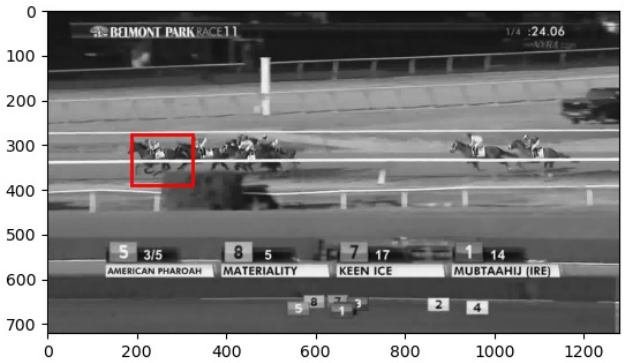

0
200
400
600
800
0
200
400
600
800
1000

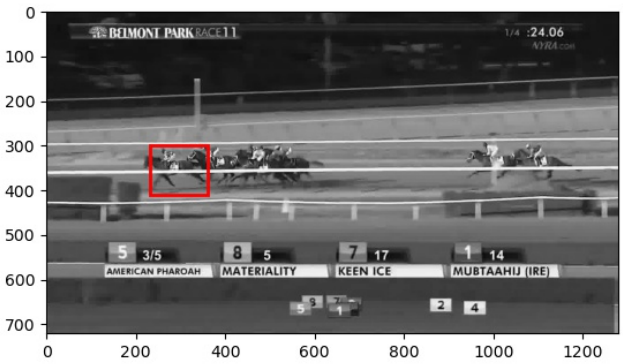

0
200
400
600
800
0
200
400
600
800
1000

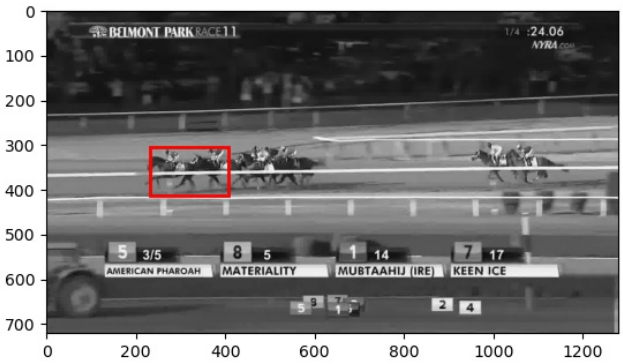

0
200
400
600
800
0
200
400
600
800
1000

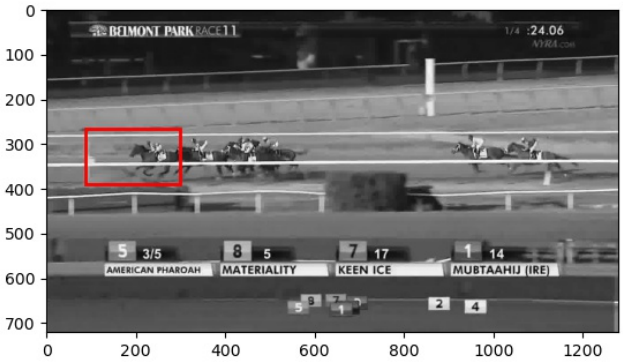
Race:


BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

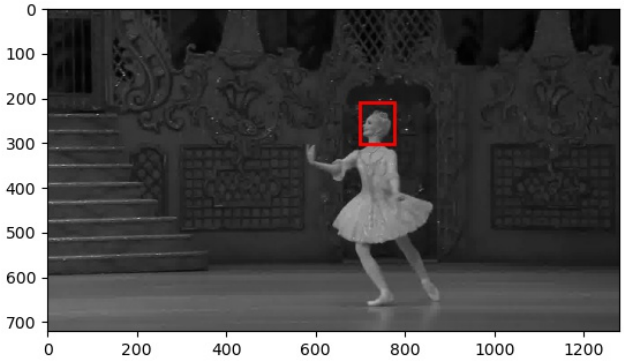

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

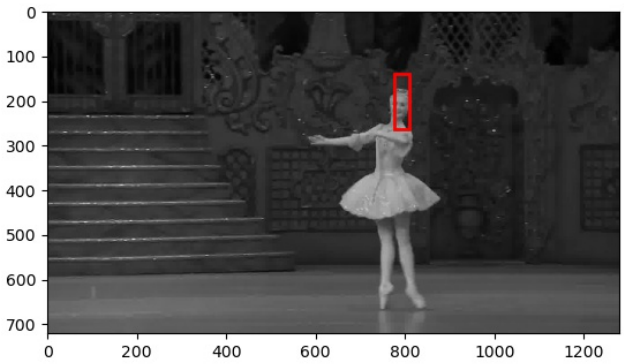

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

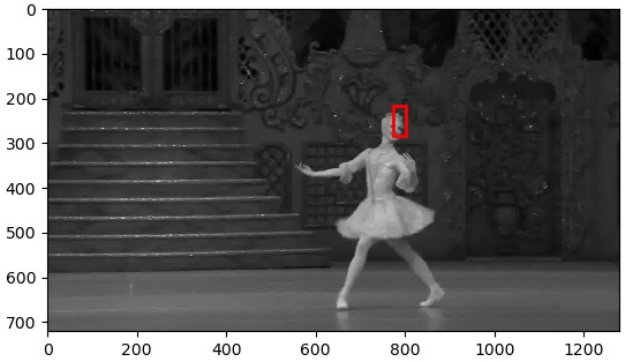

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE

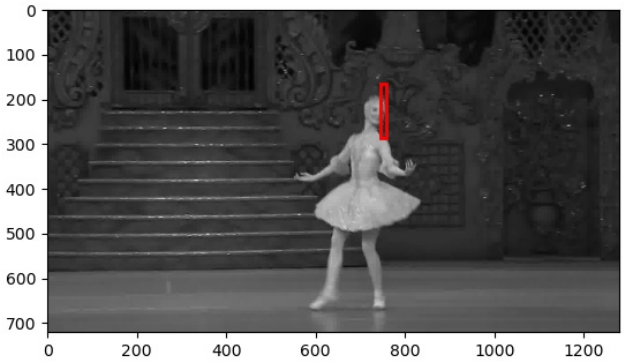

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

Ballet (only to frame 15, got negative bounding box values afterwards):




0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

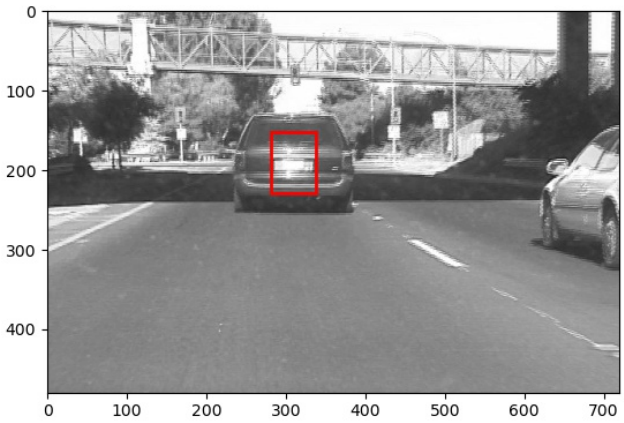

0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

Matthew-Bakers Inverse Compositional Alignment with Affine

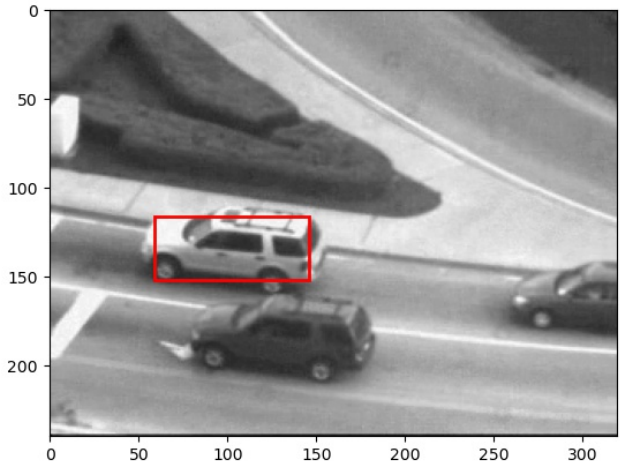
Car1 (only to frame 169, got negative bounding box values afterwards):

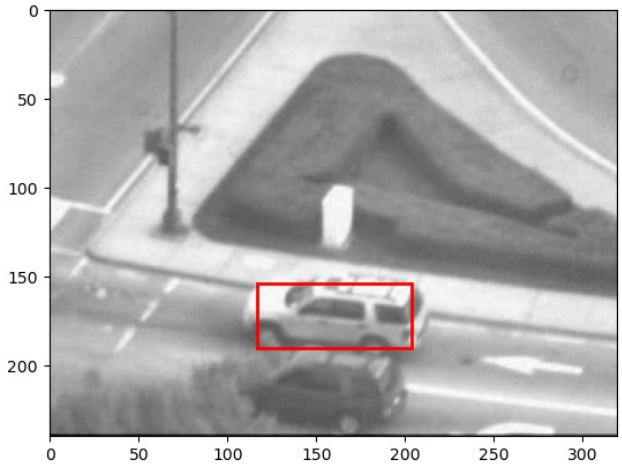

0
100
200
300
400
0
100
200
300
400
500
600
700


0
100
200
300
400
0
100
200
300
400
500
600
700

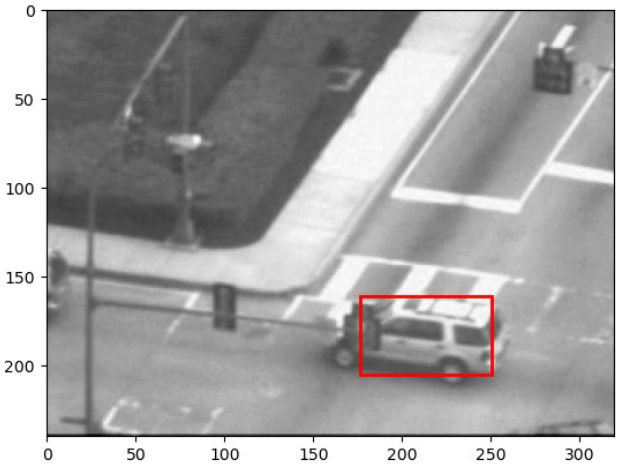

0
100
200
300
400
0
100
200
300
400
500
600
700

Car2 (only to frame 166, results not ideal afterwards):



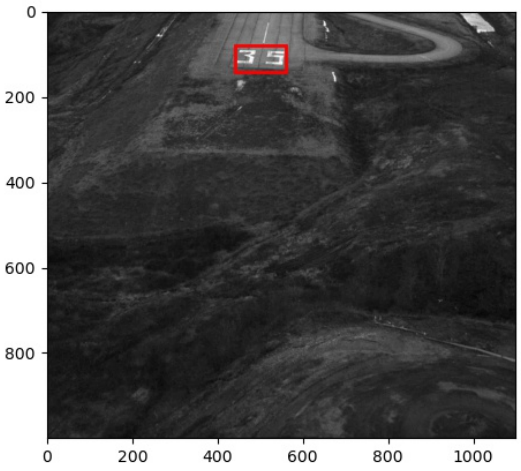


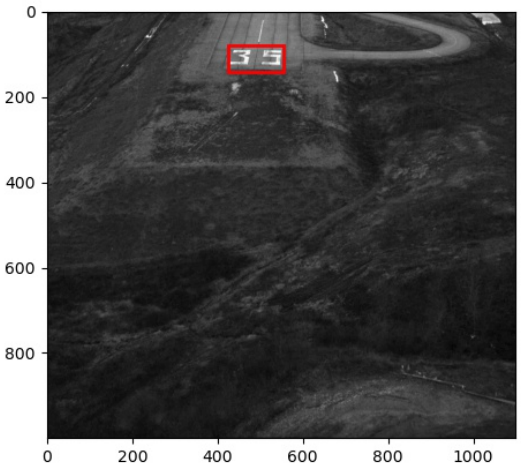

0
50
100
150
200
0
50
100
150
200
250
300

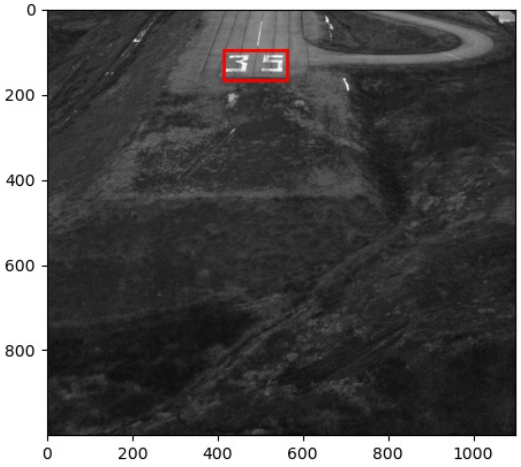

0
50
100
150
200
0
50
100
150
200
250
300

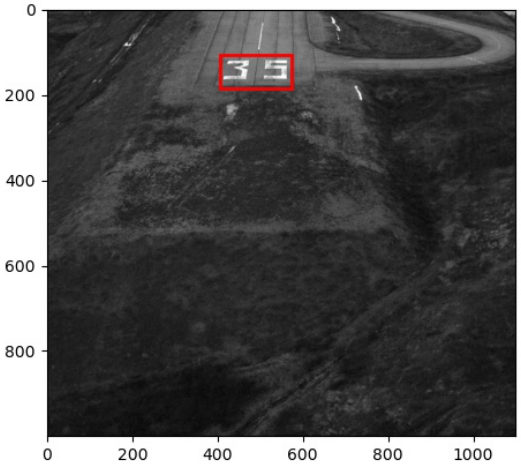

0
50
100
150
200
0
50
100
150
200
250
300

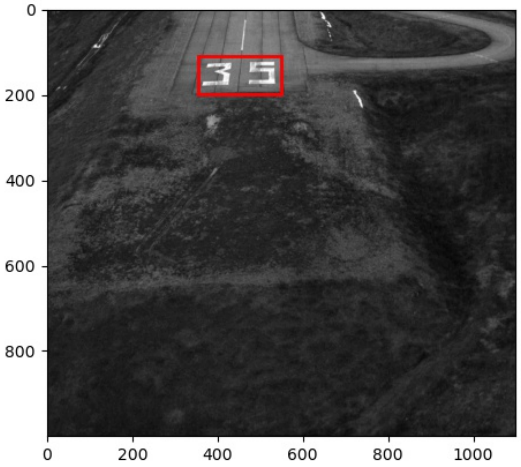
Landing:


0
200
400
600
800
0
200
400
600
800
1000

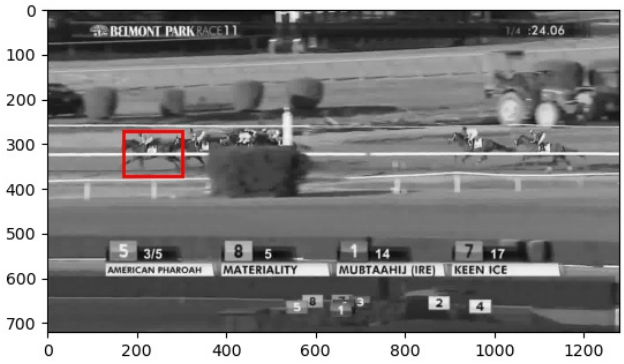

0
200
400
600
800
0
200
400
600
800
1000

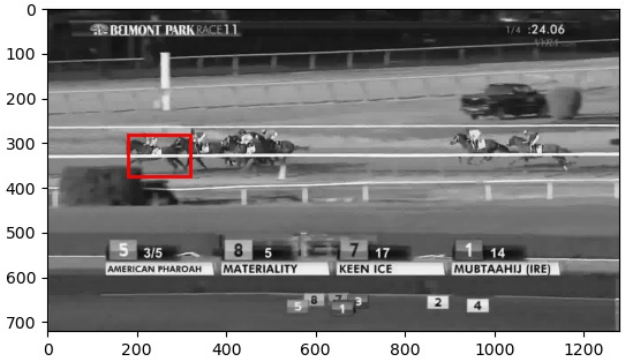

0
200
400
600
800
0
200
400
600
800
1000

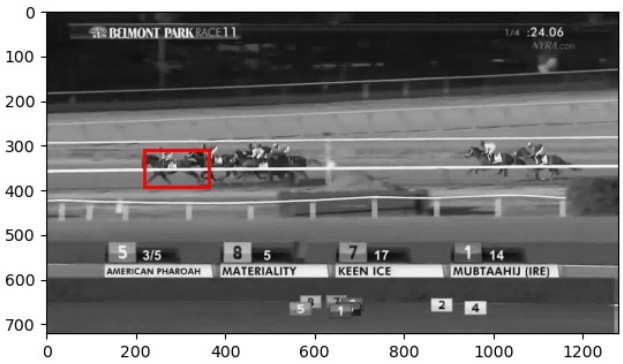

0
200
400
600
800
0
200
400
600
800
1000

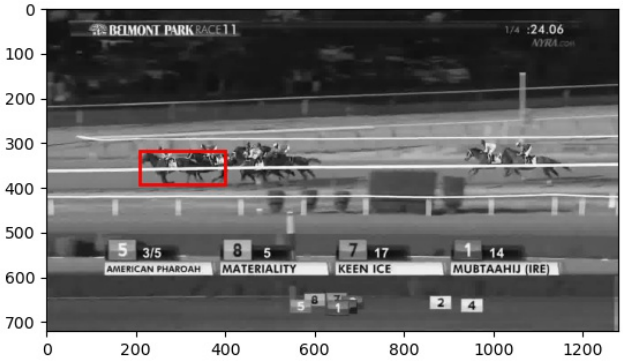

0
200
400
600
800
0
200
400
600
800
1000

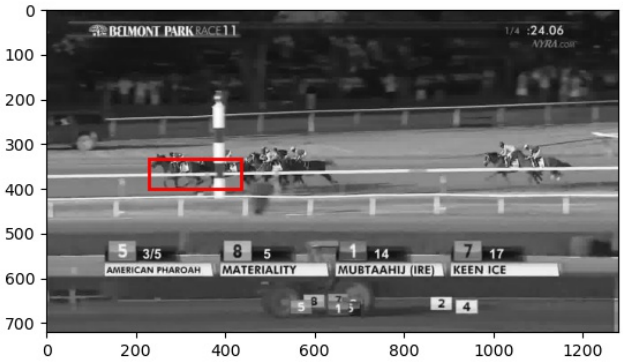
Race:


BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

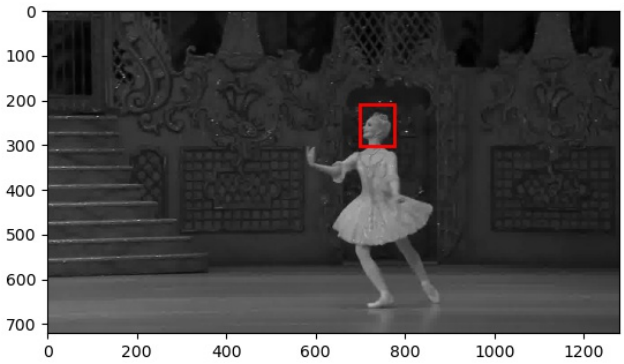

BELMONT PARK RACE 11
1/4 :24.06
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)
0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

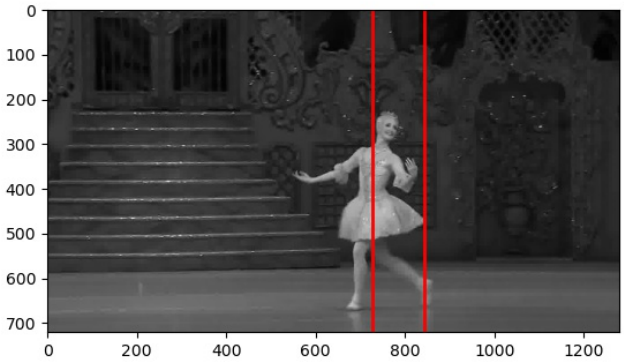

BELMONT PARK RACE 11
1/4 :24.06
NYRA.com
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

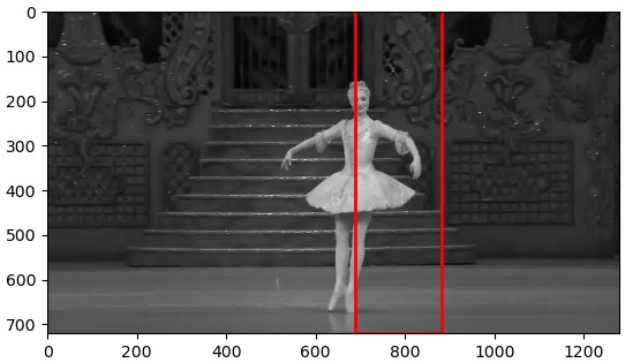

BELMONT PARK RACE 11
1/4 :24.06
NYRA.com
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
7 17
KEEN ICE
1 14
MUBTAAHIJ (IRE)

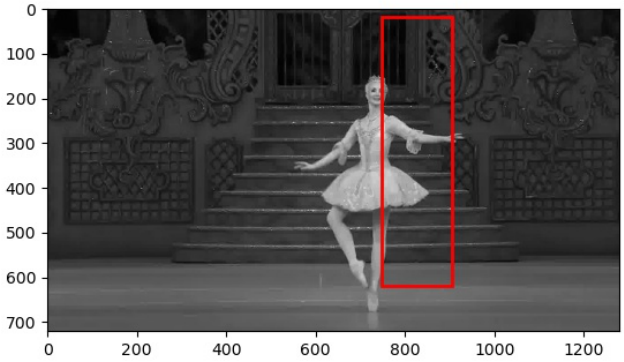

BELMONT PARK RACE 11
1/4 :24.06
NYRA.com
5 3/5
AMERICAN PHAROAH
8 5
MATERIALITY
1 14
MUBTAAHIJ (IRE)
7 17
KEEN ICE

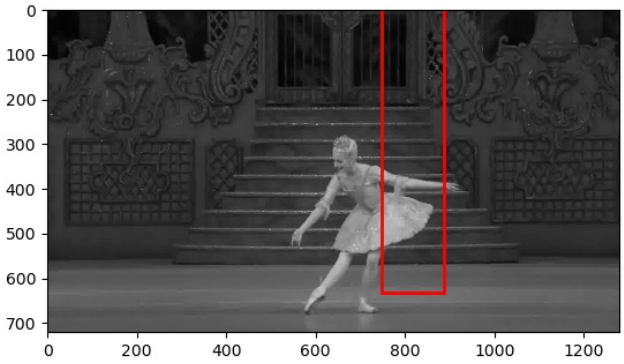
Ballet (the results are not ideal for this video):






0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

All three algorithms seem to work well on landing and race videos.

lk affine and Matthew-Bakers algorithm work well on car2 video initially,
but when the car meets the traffic light, the bounding box will extend,
following the traffic light.
I think this is because the the program starts to
consider the traffic light as part of the object being tracked.

lk affine and Matthew-Bakers algorithm does not work well for ballet video,
sometimes I got negative bounding box coordinates.

The lk translation algorithm is generally accurate,
but the size of the bounding box won't change (because we only consider translation).

The lk affine algorithm is slow,
because we need to compute the hessian and jacobian matrix for each iteration.
The size of the bounding box can change with this algorithm.

The Matthew-Bakers algorithm gives similar result as lk affine,
but it is faster because the hessian and jacobian are precomputed.

3 Extra Credit

Q3.1x

I did not do this question.

Q3.2x

I did not do this question.

Q3.3x

I (jiaqigen) hosted two study group sessions with Qichen Fu (qichenf).
Here are the screen shots.