Final Project

Matias Berretta

CISC 6525 Artificial Intelligence
Professor Damian Lyons

December 2018

Matias Berretta
Final Project

OBJECTIVE

Goal

The objective of my project is to compare the performance of the Kalman Filter to Particle Filtering on a computer vision task which consists of tracking a blue ball across time and space in a video.

Context: Recursive Bayesian Estimations

Both the Kalman Filter and the Particle Filter are Recursive Bayesian Estimations. However, the Kalman Filter is generally used for linear systems with Gaussian noise, whereas the Particle Filter is usually applied to systems involving elements of nonlinearity and non-Gaussianity.

Kalman Filtering

Kalman filtering is an algorithm that allows us to produce estimates of unknown variables by combining the information of various measurements over time via joint distribution probability. At each time frame, it will calculate the joint probability distribution over all the variables taken into account, thus negating the noise and inaccuracies they each contain on their own.

Particle Filtering

Particle filters are sequential Monte Carlos methods based on point mass (or "particle") representations of probability densities. Particle filter uses random sampling to generate different systems states, assigning bigger weights to the states whose likelihood is corroborated by sensor data. Conversely, particles whose weight falls below a certain threshold are discarded, at which point new random samples are generated to keep the number of particles constant.

METHODS

Technology

For this project I am using Anaconda Inc.'s Distribution of Python 3 (version 3.6.5). Furthermore, I am using classic python libraries such as Pandas (version 0.23.0) and NumPy (version 1.14.3) for data manipulation, as well as Matplotlib (version 2.2.2) and Seaborn version (0.9.0) for data visualization.

Apart from using OpenCV (version 3.4.1), the classic computer vision library, as the basic framework for all of my code, I have also incorporated two additional computer vision libraries to get my filters up and running: (1) I used *pykalman* to implement my Kalman Filter; (2) and *deepgaze* to implement my Particle Filter. When using anaconda's distribution of python, installing the former (1) is pretty straight-forward:

```
conda install -c conda-forge pykalmam
```

However, installing the latter (2) requires a bit more lines of code:

```
git clone https://github.com/mpatacchiola/deepgaze.git
cd deepgaze
sudo python setup.py install
sudo python setup.py install --record record.txt
```

Furthermore, I used OpenCV's CSRT tracker to set the ground truth for the ball's trajectory, against which both the Kalman and Particle Filtering would be compared.

Evaluation

Both my Kalman Filter to my Particle Filter generate estimations of where the target ball will be in space (via cartesian coordinates) at any given frame. In order to evaluate each filter's performance I compare their predictions $\hat{y}$, against the true position of the target y, with an evaluation function, $\mathcal{L}(\hat{y},\ y)$. Given the continuous and two-dimensional spatial nature of my target variable, wherein each instance consists of a pair of x and y coordinates, I use Euclidian Distance as my evaluation metric, where the lower the score, the better the filter's performance.

The objective of this project—to compare the Kalman Filter to the Particle Filter—make it clear where I should draw my prediction $\hat{y}$ from. However, in order to evaluate their performance, I needed to provide a ground truth y against which I could compare each filter's $\hat{y}$. The task of finding a source of ground truth was left to my discretion. I found the idea of manually providing coordinates frame by frame to be too time-consuming and inexact. Instead, I opted for using one of OpenCV's many Object Trackers: the CSRT tracker is a Discriminative Correlation Filter Tracker with Channel and Spatial Reliability that far outperforms both the Kalman and Particle Filters when it comes to object tracking tasks.

In short, I used OpenCV's CSRT tracker to determine the true position of the ball, target y; and I drew my predictions $\hat{y}$ from the output of the Kalman Filter and the Particle Filter. (Target y and prediction $\hat{y}$ are machine learning terms that refer to vectors and should not be confused with the cartesian plane y, which indicates an object's position on a vertical scale and acts as a counterpart to the cartesian plane x.)

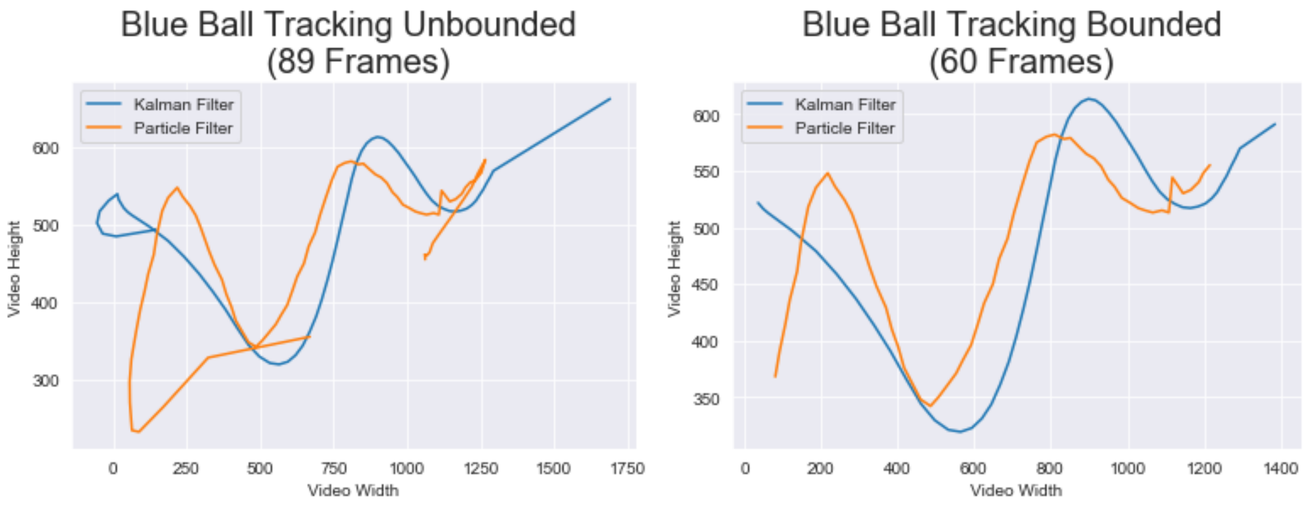

Figure 1: Unbounded and Bounded Tracking

Furthermore, it is worth noting that I cut down the number of frames in the video [Figure 1: Blue Ball Tracking Bounded], to ensure that the ball was always clearly visible within the frame of the video. In other words, I made sure that within the scope of my evaluation (60 frames, from frame 10 to frame 70) there was no instance wherein the ball was outside the frame or clipped by the frame, since this would produce strange off-shoots for both the Kalman and Particle Filters [Figure 1: Blue Ball Tracking Unbounded]. I do not consider these off-shoots to be a fair reflection of tracking performance. Moreover, the latter would interfere with my evaluation method: the CSRT tracker I used to set the ground truth trajectory can only be initialized at a moment when the ball is fully visible, given that the object to be tracked is defined by a bounding box which needs to encompass the object fully.

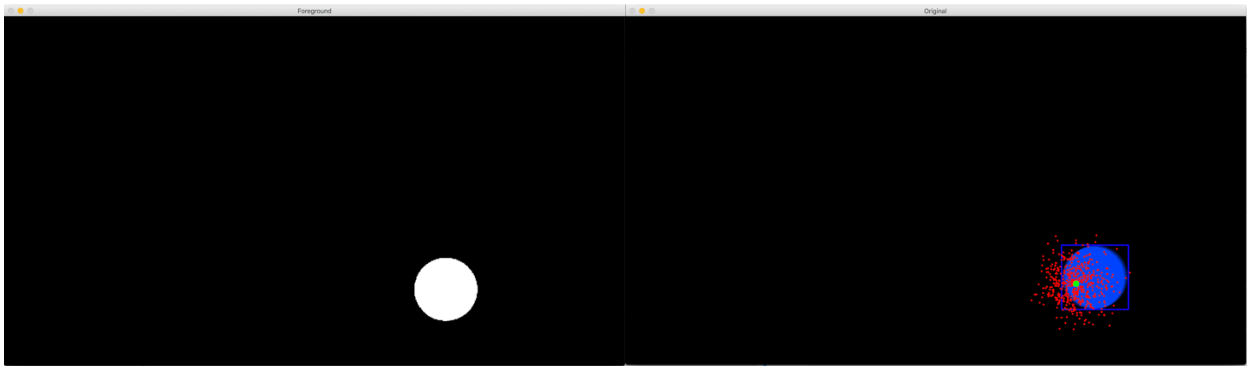

Figure 2: Filters in action. Left: Contrast mask for Kalman Filter. Right: Particle Filter in Action**;** red dots represent the particles, and the green dot represents their center.

RESULTS

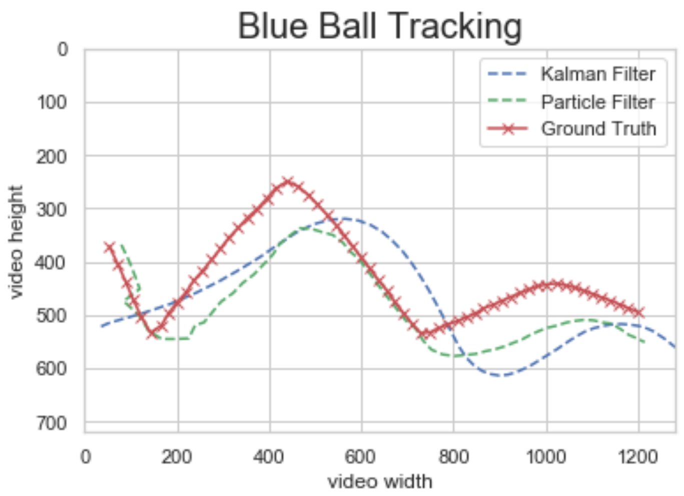

Figure 3: Blue Ball Tracking. Shows the blue ball's trajectory across 60 frames as determined by the Kalman Filter, the Particle Filter and the Ground Truth (CSRT Tracker).

Filter	Average Euclidean Distance From Ground Truth
Kalman Filter	**255.69**
Particle Filter	**60.68**

As expected, the Particle Filter outperformed the Kalman Filter, with a mean Euclidian Distance score of 60.68 vs 255.69 [Table 1]. I say, "expected", because it is a well-documented fact that the Kalman Filter's assumptions of linearity and Gaussianity limits its ability to adapt to quick, unexpected changes in the trajectory of an object. That being said, given the fairly smooth and wave-like trajectory of the ball in the video, I did not expect the difference to be so pronounced. Alas, it seems the Kalman Filter is considerably worse than the Particle Filter at adapting to sudden changes in direction, even when these changes are not particularly drastic or irregular.

For my evaluation metric I used the mean Euclidean Distance across 60 frames of the video where the ball was fully visible and contained within the physical limits of the screen. Figure 3 shows the trajectory of the ball within these 60 frames as described by the three different tracking methods employed in this project: (1) The Kalman Filter, (2) The Particle Filter, and (3) and CSRT tracker, which sets the Ground Truth.

Figure 4, acts as a counterpart to Table 1, which only offers a summary statistic of the filter's performance, by showing both filters' performance frame by frame. It is easy to see how quickly the Kalman Filter veered off the mark (especially from frame 5 to 25), most likely due to the unexpected change in direction from the ball. In contrast, the Particle Filter seems to have adjusted pretty well to these changes in directions. Still, it is worth nothing the bumps at frame 15 and 45, which demonstrate both filters struggled the most whenever the ball veered into a new direction.

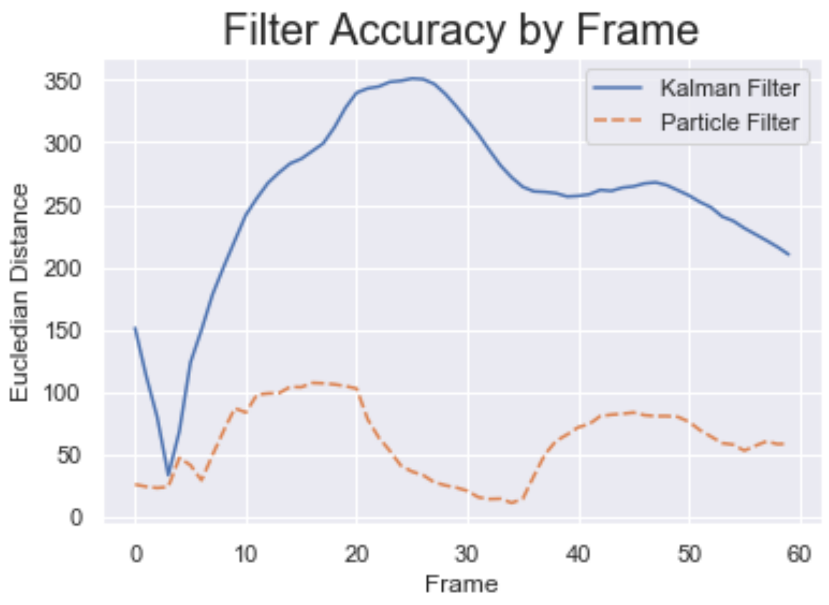

Figure 4: Filter Accuracy by Frame

Given that the Particle Filter outperformed the Kalman Filter at virtually every frame, it seems that modelling your environment without any assumptions of linearity and Gaussianity produces more accurate tracking in general, since the filter can quickly adapt to sudden changes in direction. Whereas the Kalman Filter may work perfectly fine for tracking something big and steady, i.e. a satellite or an airplane, I would argue it would struggle to properly track the movement of something smaller and prone to sudden movements, i.e. a bird or a rabbit. (I am aware that these filters can be used to track more abstract entities like the economy.) In defense of the Kalman Filter, it is worth pointing out that it is less computationally expensive than the Particle Filter.

The Kalman Filter is good fit for linear state transitions and Gaussian system noise. However, when it comes to arbitrary distributions, the Particle Filter far outshines the Kalman Filter because it uses random particles to represent the probability distribution instead of means and covariances. The more samples, the better the estimate, especially because all these samples are weighted appropriately by the Particle Filter’s sensor information.

APPENDIX

I wrote three coding scripts:

- Kalman Filter (kalman.py)
    - Ran Kalman Filter on Blue Ball Video
    - Saved coordinates for filter trajectory
- Particle Filter (particle.py)
    - Ran Particle Filter on Blue Ball Video
    - Saved coordinates for filter trajectory
- Ground Truth (ground_truth.py)
    - Ran CSRT Tracker on Blue Ball Video
    - Saved coordinates for tracker trajectory
- Evaluation & Visualization (evaluation_visualization.py)
    - Imported coordinates for all three trajectories mentioned above
    - Plotted trajectories against each other
    - Evaluated Kalman Filter and Particle Filter trajectories against Ground Truth Trajectory using Euclidean Distance.
    - Plotted Euclidean Distance from Ground Truth frame by frame.

kalman.py

```
# Download necessary libraries
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pykalman import KalmanFilter

# Set filename
filename ="BlueBounce.mp4"

# Capture video
video = cv2.VideoCapture(filename)
video.set(1,10)

# start at a point where the ball is already entirely visibile for fair comparison with ground truth
# frame number 10 will do
#video.set(1,10)

# get video height and width to set appropriate plot axes
video_width = video.get(cv2.CAP_PROP_FRAME_WIDTH)
video_height = video.get(cv2.CAP_PROP_FRAME_HEIGHT)
numframes = video.get(7)

# Set Model Parameters
count = 0
history = 15
nGauss = 3
bgThresh = 0.5
noise = 30
bgs = cv2.bgsegm.createBackgroundSubtractorMOG(history,nGauss,bgThresh,noise)

ball_trajectory = np.zeros((int(numframes), 2)) - 1

while count < numframes:

    count += 1

    img2 = video.read()[1]

    try:
        cv2.imshow('Video', img2) # some times will generate error -- ignore
    except:
        pass

    # apply background subtractor
    formatting = bgs.apply(img2)

    # set threshold for black and white masking
    ret,threshold = cv2.threshold(formatting,127,255,0)

    #_ is added to account for update in OpenCV's findContours function
    # which now returns 3 rather than 2 elements

    _, contours, hierarchy = cv2.findContours(threshold,
cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    len_contours = len(contours)

    if len_contours > 0:

        # find center of the ball (contour)
        center = np.mean(contours[0], axis = 0)
        ball_trajectory[count - 1, :] = center[0]

    try:
        cv2.imshow('Foreground',formatting) # some times will generate error -- ignore
    except:
        pass
    cv2.waitKey(80)

video.release()

measured = ball_trajectory

while True:

    if measured[0,0] == -1.:
        measured = np.delete(measured, 0, 0)
    else:

        break

numMeas = measured.shape[0]

# new section explain
marked_measure = np.ma.masked_less(measured,0)

transition_matrix=[[1,0,1,0],[0,1,0,1],[0,0,1,0],[0,0,0,1]]
observation_matrix=[[1,0,0,0],[0,1,0,0]]

X_init = marked_measure[0,0]
Y_init = marked_measure[0,1]
VX_init = marked_measure[1,0] - marked_measure[0,0]
VY_init = marked_measure[1,1] - marked_measure[0,1]
```

```
init_state = [X_init, Y_init, VX_init, VY_init]

covariance_init = 1.0e-3 * np.eye(4)
covariance_transit = 1.0e-4 * np.eye(4)
covariance_observed = 1.0e-1 * np.eye(2)

kf = KalmanFilter(transition_matrices= transition_matrix,
        observation_matrices = observation_matrix,
        initial_state_mean = init_state,
        initial_state_covariance = covariance_init,
        transition_covariance = covariance_transit,
        observation_covariance = covariance_observed)

# Refine with Kalman Filter using pykalman
(filtered_state_means, filtered_state_covariances) = kf.filter(marked_measure)

kalmanDF = pd.DataFrame({'x':filtered_state_means[:,0], 'y':filtered_state_means[:,1]})
#kalmanDF.to_csv('kalman_unbounded.csv') # unbounded kalman trajectory
len(kalmanDF)

kdf = kalmanDF.iloc[9:-20,:].reset_index().iloc[:,1:]
#kdf.to_csv('kalman_trajectory.csv') # clipped kalman trajcetory
len(kdf)

plt.axis([0,video_width ,video_height,0])
plt.plot(kdf['x'], kdf['y'], '-b' ,label = 'kalman prediction')
plt.legend(loc = 1)
plt.title("Constant Velocity Kalman Filter")
plt.show()
```

particle.py

```
import cv2
import numpy as np
from deepgaze.color_detection import BackProjectionColorDetector, RangeColorDetector
from deepgaze.mask_analysis import BinaryMaskAnalyser
from deepgaze.motion_tracking import ParticleFilter
import matplotlib.pyplot as plt
import pandas as pd

# Set filename
filename = "BlueBounce.mp4"

# Set template (to identify object, i.e. blue ball)
template = cv2.imread('BlueBounceTemplate.png') #Load the image

# Capture video
video = cv2.VideoCapture(filename)
#video.set(1,10)

# Get video frame
video_width = video.get(cv2.CAP_PROP_FRAME_WIDTH)
video_height = video.get(cv2.CAP_PROP_FRAME_HEIGHT)

#Declaring the binary mask analyser object
my_mask_analyser = BinaryMaskAnalyser()

# Defining the deepgaze color detector object
my_back_detector = BackProjectionColorDetector()
my_back_detector.setTemplate(template)

# Filter parameters
n_particles = 500 # set number of particles

# Probability of error (a.k.a. added noise)
noise_probability = 0.015

# Spread of the particles in prediction phase
std = 25

# Initialize model within video frame
tracking_particles = ParticleFilter(video_width, video_height, n_particles)

# Had to adjust the following three deepgaze functions manually to account for new update in
OpenCV 3.4.1
# (deepgaze library is built on top of OpenCV)
# wherein cv2.findCountours returned 3 values rather than 2 values
# I made the same adjustment for all three functions
```

```
def max_area_rectangle(mask, color=[0, 0, 255]):
        """it returns the rectangle sorrounding the contour with the largest area.

        This method could be useful to find a face when a skin detector filter is used.
        @param mask the binary image to use in the function
        @return get the coords of the upper corner of the rectangle (x, y) and the rectangle size
(widht, hight)
           In case of error it returns a tuple (None, None, None, None)
        """
        if(mask is None): return (None, None, None, None)
        mask = np.copy(mask)
        if(len(mask.shape) == 3):
           mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
        _, contours, hierarchy = cv2.findContours(mask, 1, 2)
        area_array = np.zeros(len(contours)) #contains the area of the contours
        counter = 0
        for cnt in contours:
              area_array[counter] = cv2.contourArea(cnt)
              counter += 1
        if(area_array.size==0):
           return (None, None, None, None) #the array is empty
        max_area_index = np.argmax(area_array) #return the index of the max_area element
        cnt = contours[max_area_index]
        (x, y, w, h) = cv2.boundingRect(cnt)
        return (x, y, w, h)

def total_contour(mask, color=[0, 0, 255]):
        """it returns the total number of contours present on the mask

        this method must be used during video analysis to check if the frame contains
        at least one contour before calling the other function below.
        @param mask the binary image to use in the function
        @return get the number of contours
        """
        if(mask is None):
           return None
        mask = np.copy(mask)
        if(len(mask.shape) == 3):
           mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
        _, contours, hierarchy = cv2.findContours(mask, 1, 2)
        if(hierarchy is None):
           return 0
        else: return len(hierarchy)

def max_area_center(mask, color=[0, 0, 255]):
        """it returns the centre of the contour with largest area.
        This method could be useful to find the center of a face when a skin detector filter is used.
        @param mask the binary image to use in the function
        @return get the x and y center coords of the contour whit the largest area.
          In case of error it returns a tuple (None, None)
        """
        if(mask is None): return (None, None)
        mask = np.copy(mask)
        if(len(mask.shape) == 3):
          mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
        _, contours, hierarchy = cv2.findContours(mask, 1, 2)
        area_array = np.zeros(len(contours)) #contains the area of the contours
        counter = 0
        for cnt in contours:
            #cv2.drawContours(image, [cnt], 0, (0,255,0), 3)
            #print("Area: " + str(cv2.contourArea(cnt)))
            area_array[counter] = cv2.contourArea(cnt)
            counter += 1
        if(area_array.size==0):
          return (None, None) #the array is empty
        max_area_index = np.argmax(area_array) #return the index of the max_area element
        #Get the centre of the max_area element
        cnt = contours[max_area_index]
        M = cv2.moments(cnt) #calculate the moments
        if(M['m00'] == 0):
          return (None, None)
        cx = int(M['m10']/M['m00']) #get the center from the moments
        cy = int(M['m01']/M['m00'])
        return (cx, cy) #return the center coords

trajectory_measurements_x = []
trajectory_measurements_y = []
try:
    while(True):

        # Capture frame-by-frame
        ret, frame = video.read()
        if(frame is None):
          break #check for empty frames

        #Return the binary mask from the backprojection algorithm
        mask = my_back_detector.returnMask(frame, morph_opening=True, blur=True,
kernel_size=5, iterations=2)
        frame_mask = cv2.bitwise_not(mask) # invert mask so that it tracks ball and not
background


        if(total_contour(frame_mask) > 0):
          # Use the binary mask to find the contour with largest area
```

```
            # then find the center of this contour (what we want to track)
            x_rect,y_rect,w_rect,h_rect = max_area_rectangle(frame_mask)
            x_center, y_center = max_area_center(frame_mask)

            #Add noise
            coin = np.random.uniform()
            if(coin >= 1.0-noise_probability):
                x_noise = int(np.random.uniform(-300, 300))
                y_noise = int(np.random.uniform(-300, 300))
            else:
                x_noise = 0
                y_noise = 0
            x_rect += x_noise
            y_rect += y_noise
            x_center += x_noise
            y_center += y_noise
            cv2.rectangle(frame, (x_rect,y_rect), (x_rect+w_rect,y_rect+h_rect), [255,0,0], 2) # blue
rectangle

        # Predict target's position
        tracking_particles.predict(x_velocity=0, y_velocity=0, std=std)

        # Draw the particles
        tracking_particles.drawParticles(frame)

        # Estimate the object's position in the next frame
        x_estimated, y_estimated, _, _ = tracking_particles.estimate()
        cv2.circle(frame, (x_estimated, y_estimated), 3, [0,255,0], 5) # green dot

        # Save trajectory coordiantes frame by frame
        trajectory_measurements_x.append(x_estimated)
        trajectory_measurements_y.append(y_estimated)

        # Update the filter with the last measurements
        tracking_particles.update(x_center, y_center)

        #Resample the particles
        tracking_particles.resample()

        #Show the frame and wait for the exit command
        cv2.imshow('Original', frame) #show on window
        cv2.imshow('Mask', frame_mask) #show on window
        if cv2.waitKey(1) & 0xFF == ord('q'): break #Exit when Q is pressed
except: # sometimes thi part of the code needs to be run twice to work
    while(True):

        # Capture frame-by-frame
        ret, frame = video.read()
    if(frame is None):
      break #check for empty frames

    #Return the binary mask from the backprojection algorithm
    mask = my_back_detector.returnMask(frame, morph_opening=True, blur=True,
kernel_size=5, iterations=2)
    frame_mask = cv2.bitwise_not(mask) # invert mask so that it tracks ball and not
background


    if(total_contour(frame_mask) > 0):
      # Use the binary mask to find the contour with largest area
      # then find the center of this contour (what we want to track)
      x_rect,y_rect,w_rect,h_rect = max_area_rectangle(frame_mask)
      x_center, y_center = max_area_center(frame_mask)

      #Add noise
      coin = np.random.uniform()
      if(coin >= 1.0-noise_probability):
        x_noise = int(np.random.uniform(-300, 300))
        y_noise = int(np.random.uniform(-300, 300))
      else:
        x_noise = 0
        y_noise = 0
      x_rect += x_noise
      y_rect += y_noise
      x_center += x_noise
      y_center += y_noise
      cv2.rectangle(frame, (x_rect,y_rect), (x_rect+w_rect,y_rect+h_rect), [255,0,0], 2) # blue
rectangle

    # Predict target's position
    tracking_particles.predict(x_velocity=0, y_velocity=0, std=std)

    # Draw the particles
    tracking_particles.drawParticles(frame)

    # Estimate the object's position in the next frame
    x_estimated, y_estimated, _, _ = tracking_particles.estimate()
    cv2.circle(frame, (x_estimated, y_estimated), 3, [0,255,0], 5) # green dot

    # Save trajectory coordiantes frame by frame
    trajectory_measurements_x.append(x_estimated)
    trajectory_measurements_y.append(y_estimated)

    # Update the filter with the last measurements
    tracking_particles.update(x_center, y_center)
    #Resample the particles
    tracking_particles.resample()

    #Show the frame and wait for the exit command
    cv2.imshow('Original', frame) #show on window
    cv2.imshow('Mask', frame_mask) #show on window
    if cv2.waitKey(1) & 0xFF == ord('q'): break #Exit when Q is pressed


#Release the camera
video.release()

# save x and y coordinates of particle filter trajectory into pandas dataframe
unbounded_particleDF = pd.DataFrame({"x":trajectory_measurements_x,
"y":trajectory_measurements_y})
#unbounded_particleDF.to_csv('unbounded_particle.csv') # unbounded particle trajectory

particleDF = pd.DataFrame({"x":trajectory_measurements_x[9:-20],
"y":trajectory_measurements_y[9:-20]})
len(particleDF) # 60 frames
#particleDF.to_csv('particle_trajectory.csv') # clipped particle trajectory
# save x and y coordinates of particle filter trajectory into pandas dataframe
plt.figure()
plt.axis([0,video_width,video_height, 0])
plt.plot(particleDF['x'], particleDF['y'], '-g')
```

ground_truth.py

```
import cv2
import pandas as pd
import matplotlib.pyplot as plt

tracker = cv2.TrackerCSRT_create()

tracking_box = [] # track x and y coordinates for ground truth comparsion

# Read video
video = cv2.VideoCapture("BlueBounce.mp4")
video.set(1,9)
# Exit if video not opened.
if not video.isOpened():
    print("Could not open video")
    sys.exit()

# Read first frame.
ok, frame = video.read()
if not ok:
    print('Cannot read video file')
    sys.exit()

# Define an initial bounding box
bbox = (35, 338, 130, 130)

# Uncomment the line below to select a different bounding box
# bbox = cv2.selectROI(frame, False)

#print(bbox)

# Initialize tracker with first frame and bounding box
ok = tracker.init(frame, bbox)

while True:
    # Read a new frame
    ok, frame = video.read()
    if not ok:
        break

    # Start timer
    timer = cv2.getTickCount()

    # Update tracker
    ok, bbox = tracker.update(frame)
    print(bbox[0:2])
    # track progress
    tracking_box.append(bbox)

    # Calculate Frames per second (FPS)
    fps = cv2.getTickFrequency() / (cv2.getTickCount() - timer);

    # Draw bounding box
    if ok:
            # Tracking success
        p1 = (int(bbox[0]), int(bbox[1]))
        p2 = (int(bbox[0] + bbox[2]), int(bbox[1] + bbox[3]))
        cv2.rectangle(frame, p1, p2, (255,0,0), 2, 1)
    else :
        # Tracking failure
        cv2.putText(frame, "Tracking failure detected", (100,80),
cv2.FONT_HERSHEY_SIMPLEX, 0.75,(0,0,255),2)

    # Display tracker type on frame
    cv2.putText(frame, "CSRT" + " Tracker", (100,20), cv2.FONT_HERSHEY_SIMPLEX, 0.75,
(50,170,50),2);

    # Display FPS on frame
    cv2.putText(frame, "FPS : " + str(int(fps)), (100,50), cv2.FONT_HERSHEY_SIMPLEX,
0.75, (50,170,50), 2);

    # Display result
    cv2.imshow("Tracking", frame)


    # Exit if ESC pressed
    k = cv2.waitKey(1) & 0xff
    if k == 27 : break

csrt_trajectory = tracking_box[:-20] # keep it at 60 frames
csrtDF = pd.DataFrame(csrt_trajectory, columns = ['x','y','drop1','drop2'])
csrtDF.drop(columns = ['drop1','drop2'], axis = 0, inplace = True)

#csrtDF.to_csv('csrt_trajectory.csv')
print(len(csrt_trajectory))# 60 frames
```

evaluation_visualization.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import math
import cv2

# Capture video and get frame measurements
filename = "BlueBounce.mp4"
video = cv2.VideoCapture(filename)
video_width = video.get(cv2.CAP_PROP_FRAME_WIDTH)
video_height = video.get(cv2.CAP_PROP_FRAME_HEIGHT)

# import trajectories for different filters + tracker
kalman = pd.read_csv('kalman_trajectory.csv', index_col = 0)
particle = pd.read_csv('particle_trajectory.csv', index_col = 0)
csrt = pd.read_csv('csrt_trajectory.csv', index_col = 0)

# unbounded trajectories
unbounded_kalman = pd.read_csv('unbounded_kalman.csv', index_col = 0)
unbounded_particle = pd.read_csv('unbounded_particle.csv', index_col = 0)

sns.set_style('darkgrid')

plt.plot(unbounded_kalman['x'], unbounded_kalman['y'])

plt.plot(unbounded_particle['x'], unbounded_particle['y'])

plt.title('Blue Ball Tracking Unbounded \n (89 Frames)', size = 20)
plt.xlabel('Video Width')
plt.ylabel('Video Height')
plt.legend(['Kalman Filter','Particle Filter'])
#plt.savefig('Filter_Accuracy_by_Frame.jpg')
plt.show()

sns.set_style('darkgrid')

plt.plot(unbounded_kalman['x'][9:-20], unbounded_kalman['y'][9:-20])

plt.plot(unbounded_particle['x'][9:-20], unbounded_particle['y'][9:-20])

plt.title('Blue Ball Tracking Bounded \n (60 Frames)', size = 20)
plt.xlabel('Video Width')
plt.ylabel('Video Height')
plt.legend(['Kalman Filter','Particle Filter'])
#plt.savefig('Filter_Accuracy_by_Frame.jpg')
```

```
plt.show()

def eucledian_distance(df1, df2):

    """
    Takes in two sets of coordinates and calculates the eucledian distance between them
    returns a list reflecting the eucledian point at each distance
    Used to determine how close each filter is to the ground truth.
    """

    distances = []

    for i in range(len(df1)):

        x = df1['x'][i]
        y = df1['y'][i]
        a = df2['x'][i]
        b = df2['y'][i]

        dist = math.sqrt(math.pow(np.abs(x-a),2)+math.pow(np.abs(y-b),2))
        distances.append(dist)

    return distances

#csrt['x_adjust'] = csrt['x'] + 60
sns.set_style('whitegrid')
plt.figure()
plt.axis([0,video_width,video_height, 0])
plt.title('Blue Ball Tracking', size = 20)
plt.xlabel('video width')
plt.ylabel('video height')
plt.plot(kalman['x'], kalman['y'], '--b')
plt.plot(particle['x'], particle['y'], '--g')
plt.plot(csrt['x'], csrt['y'], '-xr')
plt.legend(['Kalman Filter','Particle Filter', 'Ground Truth'])

particle_eval = eucledian_distance(csrt, particle)
kalman_eval = eucledian_distance(csrt, kalman)

print("Kalman Particle Mean Euclidean Score:", str(np.mean(kalman_eval)), "\nParticle Filter
Mean Euclidean Score:", str(np.mean(particle_eval)))

plt.plot(kalman_eval)
plt.plot(particle_eval)
plt.title('Filter Accuracy by Frame')
plt.xlabel('Frame')
plt.ylabel('Eucledian Distance')
plt.legend(['Kalman Filter','Particle Filter'])
plt.show()

evalDF = pd.DataFrame({"kalman":kalman_eval, "particle":particle_eval})

sns.set_style('darkgrid')
sns.lineplot(data = evalDF)
plt.title('Filter Accuracy by Frame', size = 20)
plt.xlabel('Frame')
plt.ylabel('Eucledian Distance')
plt.legend(['Kalman Filter','Particle Filter'])
#plt.savefig('Filter_Accuracy_by_Frame.jpg')
plt.show()
```