Particle Filter - Lab 7.3

Recap

This is the Lab on using a Particle Filter in CE6003's Object Tracking. You should complete the tasks in this lab as part of the Particle Filter section of the lesson.

Please remember this lab must be completed before taking the quiz at the end of this lesson.

First, if we haven't already done so, we need to clone the various images and resources needed to run these labs into our workspace.

In [1]:

```
#!git clone https://github.com/EmdaloTechnologies/CE6003.git
#!git clone https://github.com/mcnamarad1971/CE6003.git
```

Program Description

This program demonstrates a very simple 'tracking' mechanism - derived from a Particle filter. We're going to use our Particle filter to track a single object, namely a person.

In [2]:

```
import sys
#sys.path.insert(1, "/content/CE6003/code")
sys.path.insert(1, "./code")
```

In [3]:

```
import os
import re
import io
import cv2
import time
import numpy as np
import base64
from IPython.display import clear_output, Image, display
from motion_tracking import ParticleFilter  # Import pre-cooked Particle filter
```

The Story So Far

To illustrate how to track something in a video stream, we have used the following technique to generate a set of images for you to work on.

What we did was we generated a short video - just recording one person walking around, on an iPhone.

Then we used `ffmpeg` to decompose about 7 seconds of that video down into still images.

```
ffmpeg -i $vid_in -vf fps=30 imgs/daire%03d.png
```

We saved those frames as `imgs/daire%03d.png` in the git repository in the single-detections directory

We've run yolo3 over those frames to generate bounding boxes and saved those bounding boxes into the same directory.

The file format is comma-separated values and the values are as shown here:

frame index	object-type	centre-x	centre-y	width	height	confidence
int	-1	float	float	float	float	float

- The object type is always a person - that's all we inferred for.
- The centre-x and width are fractions of the image's width
- The centre-y and height are fractions of the image's height
- The confidence is supplied by Yolo3

What Happens Now

For each image in the directory, in order,

- we'll find the centre of the detection in that image (if any)
- we'll build a bounding box for the detection in that image
- we'' derive a variance term (crudely) from the Yolo confidence for that image
- and we'll supply the centre of that bounding box along with the variance term to a Particle Filter implementation

Then, we'll explore how a Particle filter tracks the object in the image stream.

Key Parameters

We have four key parameters to shape our particle filter. We have the number of particles, and three terms associated with an action estimate; xVel, yVel, and velStd

In [4]:

```
numParticles = 50
xVel = 5
yVel = 5
velStd = 25
```

Get File Handles

This function gets the filenames of all the files in the directory, in a reproducible order, and loads in the bounding boxes from file.

In [5]:

```
def get_pngs_and_boxes():
    #pngdir = "/content/CE6003/images/lab7/single-objects/"
    #bbdir = "/content/CE6003/images/lab7/single-objects/"
    pngdir = "./images/lab7/single-objects/"
    bbdir = "./images/lab7/single-objects/"

    pngfolder = os.fsencode(pngdir)
    bbfolder = os.fsencode(bbdir)

    pngfiles = []
    for filename in os.listdir(pngfolder):
        if filename.decode().endswith(".png"):
            pngfiles.append(pngdir + filename.decode())
    pngfiles.sort()

    for filename in os.listdir(bbfolder):
        if filename.decode().endswith(".boxes"):
            bbfilename = bbdir + filename.decode()

    bb = open(bbfilename, "r")
    bb_lines = bb.readlines()
    bb.close()

    return bb_lines, pngfiles
```

Parse Detections

We'll use this function in the main loop to wrangle the detections into the format we want to supply to our Particle Filter.

Essentially it takes the name of png file, an img object and the list of bounding boxes as inputs.

It then finds the correct record (if any) for that image in the bounding boxes list and converts the bounding box parameters into a format which we'll use for the rest of the program (it converts back to absolute pixel values).

It returns a centre and a confidence value for the image supplied to it.

In [6]:

```
def parse_detections(bboxes, pngfile, img):
    # Sample Line: 400,-1,0.285417,0.241667,0.094792,0.483333,0.999797,-1,-1,-1
    # Index, object type,
    # x    - centre of bounding box (as fraction of image width
    # y    - centre of bounding box (as fraction of image height
    # w    - width of bounding box (as fraction of image width)
    # h    - height of bounding box (as fraction of image height
    # prob, _,_,_

    # extract the frame index of the png file -
    # use it to find the detections for that frame
    index = int(re.findall(r'\d+', pngfile)[-1])
    imgh, imgw = img.shape[:2]

    centre = np.zeros(shape=(2, 1))
    P = 0.000001 # hack to avoid div by zero
    for line in bboxes:
        np_array = np.genfromtxt(io.StringIO(line), delimiter=",")
        lineindex = int(np_array[0])

        if lineindex == index:
            centre = np_array[2:4]
            P += np_array[6]
            centre[0] *= imgw
            centre[1] *= imgh

            return centre, P

    return centre, P
```

Particle Filter

Look back over Particle Filter for an insight into how it is operating.

The concept is:

- For a low computational cost
- Generate a set of sampled state update/prediction terms (particles)
- Generate a measurement prediction from that state
- Calculate the difference between the predicted measurement of the selected particles and the actual measurement
- Adjust the state update/prediction particles (through a resampling stage to prevent degeneration of filter), and repeat....

Done on Monte Carlo sampled particles.

One key term to watch is how many particles to use for your specific tracking application.

Demo

Program Execution

For each file:

- get centre of detection (if any) and confidence from Yolo
- feed Particle Filter with these values
- Print internal Particles used by Particle Filter

In [7]:

```
writer = None

def demo_particle():
    global writer
    global numParticles
    global velStd, xVel, yVel

    # Initialise the filter with height of the frame, width of the frame
    # and the number of particles
    particleFilter = ParticleFilter(1920, 1080, numParticles)

    bb_lines, pngfiles = get_pngs_and_boxes()

    for pngfile in pngfiles:
        #print("handling .." + os.path.basename(pngfile))
        img = cv2.imread(pngfile)

        # Derive meas-var from yolo confidence level in detection
        raw_centre, conf = parse_detections(bb_lines, pngfile, img)

        # Crudely derive meas-var.  If yolo is confident we want a small
        # uncertainty. If yolo isn't confident, translate to
        # a large uncertainty.
        if(conf > 0.50):
            lStd = velStd
        else:
            lStd = 1

        # update weights of particles based on measure
        particleFilter.update(raw_centre.item(0), raw_centre.item(1))

        # Pretty print particles
        for i in range(0, numParticles):
                x_part, y_part = particleFilter.returnParticlesCoordinates(i)
                cv2.circle(img, (x_part, y_part), 10, (0,255,0),-1)

        # Resize and show the image
        img2 = cv2.resize(img, (int(img.shape[1]/4), int(img.shape[0]/4)))

        # update model - using 5 pixels in x and y and adjusting model variance
        # depending on yolo confidence
        particleFilter.predict(x_velocity=xVel,y_velocity=yVel, std=lStd)

        # estimate the position of the point, based on particle weights
        x_est, y_est, _, _ = particleFilter.estimate()

        #The resampling draws particles from the current set with a
        # probability given by the current weights.
        # The new set is an approximation of the distribution which represents the stat
e
        # of the particles at time t.
        # The resampling solves this problem: after some iterations of the algorithm
        # some particles are useless because they do not represent the point
        #position anymore, eventually they will be too far away from the real position.
        #The resample function removes useless particles and keep the
        #useful ones.
        particleFilter.resample()

        # Build a frame of our output video
```

```
        if writer is None:
            # Initialize our video writer
            fourcc = cv2.VideoWriter_fourcc(*'VP80')
            writer = cv2.VideoWriter('video.webm', fourcc, 30, (img2.shape[1], img2.sha
pe[0]), True)

        # Write the output frame to disk
        writer.write(img2)

    # Release the file pointers
    writer.release()

demo_particle()
```

Video

Thia code plays the video we just made.

The Particles being used by the Filter dot in red.

As you can see, Particle Filtering has a role to play in predicting a reasonable guess for where the object might be while it is off-camera - it should be better than Kalman for non-linear motion.

In [8]:

```
# Set this to 1 if video display
# is not working - works with chrome and firefox, not with safari
videoBodge = 0

def arrayShow (imageArray):
    ret, png = cv2.imencode('.png', imageArray)
    encoded = base64.b64encode(png)
    return Image(data=encoded.decode('ascii'))

if(videoBodge == 0):
    from IPython.display import HTML
    from base64 import b64encode
    webm = open('video.webm','rb').read()
    data_url = "data:video/webm;base64," + b64encode(webm).decode()
else:
    video = cv2.VideoCapture("video.webm")
    while(video.isOpened()):
        clear_output(wait=True)
        ret, frame = video.read()
        if(ret == False):
          break
        lines, columns, _ =  frame.shape
        img = arrayShow(frame)
        display(img)
        time.sleep(1)
```

In [9]:

```
# Display Video
HTML("""
<video width=200 controls>
      <source src="%s" type="video/webm">
</video>
""" % data_url)
```

Out[9]:

Conclusion

Exercises

Exercise 1 Simulate occluding the object being detected - for example, only supply every second measurement update to the Particle Filter and observe the Filter behaviour.

Exercise 2 Simply multiply and divide the number of particles and observe how that affects the Particle Filter's predictions.

Takeaways

1. You've seen a Particle Filter used for single object tracking
2. You've seen that a Particle Filter can help deal with occlusions - i.e. in this example the object being tracked disappeared for a few frames and the Particle Filter continued to predict motion for it based on its model.
3. You've seen that you probably need to tune the Filter to get it working for a particular application.

Next Steps

1. We've seen a set of tracking filters, from the same 'Kalman' family. Now, we'll look at how to track multiple objects in a scene and then we'll look at other tracking techiques not in the 'Kalman' family such as Optical Flow and CNN Feature based trackers.