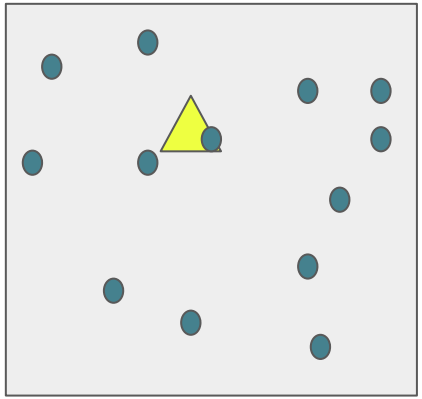
Particle Filter Based Object Tracking
2d / 3d

What are you trying to solve ?

- **Given** : input template image of target object and Input video to find target object
- **Find** : location of object in target video



S0 [t0] : Sampling
At initialization step

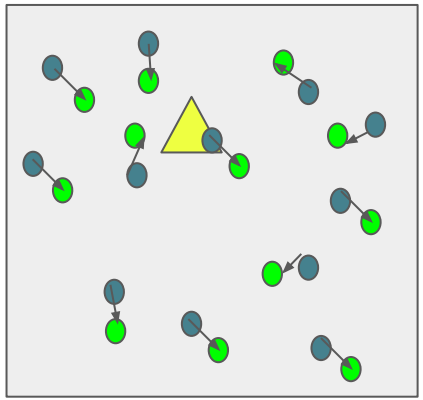
- Uniform
- random
- Prior information (incase you have)

: Target Object in Scene

: Initial Guesses of positions aka particles

```
def generateSamples():
    ### Random Initialization
    # initialise samples equally across image
    x_s = random.sample(range(10, 590), N)
    y_s = random.sample(range(10, 430), N)

    for i in range(0, N):
        s = State(x_s[i], y_s[i])
        s.udpdateWeight(1.0 / N)
        sampleList.append(s)
```



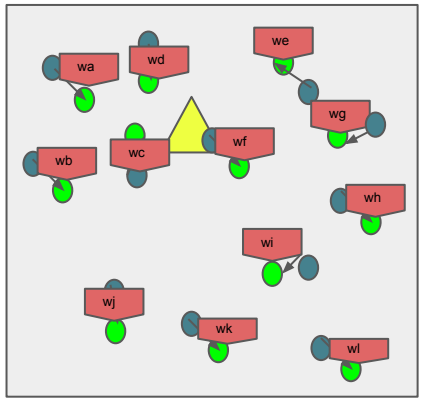
S1 [t0] : Motion Model : How we expect the object to move between frames. We used normal distribution to model this, with the assumption that the particle will move with 95% certainty within a radius of 2*std_dev pixels.
Value of std_dev would depend on how fast the object moves. Large, if object closer to camera ~40 vs Small, when object is far

: Target Object in Scene

: Initial Guesses of positions aka particles

: Particles after applying motion model

```
def applyMotionModel():
    # apply random movement motion model
    std_dev = 50
    for i in sampleList:
        cur_action = random.randint(-1, 1)  # 1 : add, 0:do nothin
        new_x = int(i.x + cur_action * std_dev * random.random())
        new_y = int(i.y + cur_action * std_dev * random.random())
```



S2 [t0]: Weight Particles : How likely each particle is at that position. To do that, we will place the template at the location of each particle and check its correspondence (or probability) of it being there.
wf : should have the highest probability, as it's closest to the target object

: Target Object in Scene

: Initial Guesses of positions aka particles

: Particles after applying motion model

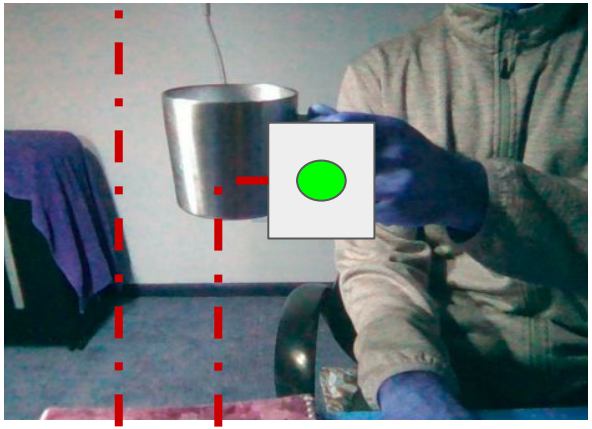
```
def weightSamples(cur_frame):                                    ▲4 ▲43 ✓20
    # weight each sample based on coherence using features
    total_liklihood = 0
    new_liklihood = 0
    for i in range(0, len(sampleList)):
        l, d = findCoherence(i, cur_frame)
        if l < 0:
            l = 0
        sampleList[i].udpdateWeight(l)
        total_liklihood += l
```

S2 [t0]: Weight Particles :
correspondence

: Target Object in Scene

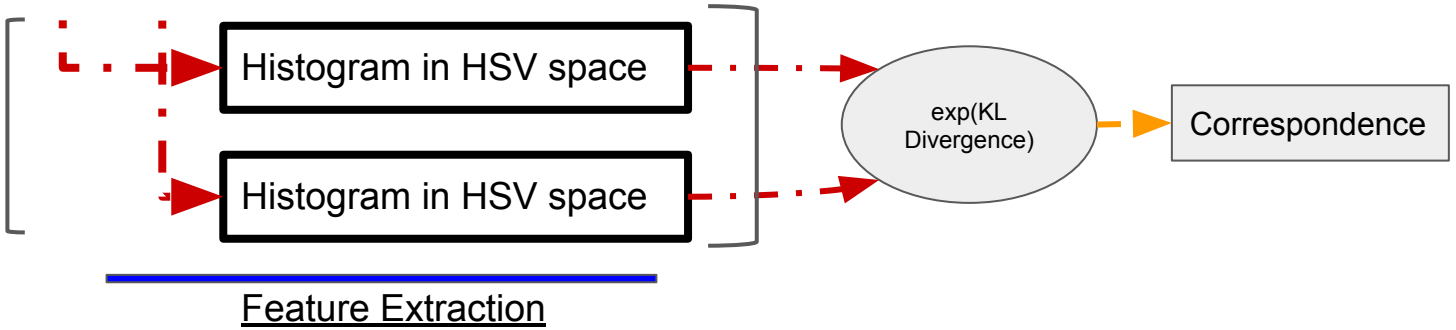
: Initial Guesses of positions aka particles

: Particles after applying motion model

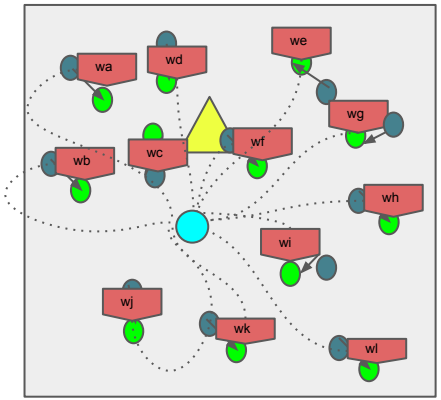


Feature Extraction :

- Histogram HSV or RGB
- SIFT Key point and Descriptor (how ?)
-



Normalize weights to turn them in probability, and then…



S3 [t0]: Predict : estimate current location, taking weighted mean of all the particles

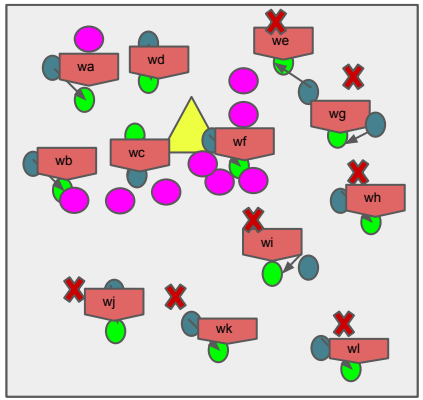
: Target Object in Scene

: Initial Guesses of positions aka particles

: Particles after applying motion model

: Predicted particle for current time

```
def predict():
    # predcit current location based on weights
    mean_x = 0
    mean_y = 0
    for k, i in enumerate(sampleList):
        mean_x += i.x * i.w
        mean_y += i.y * i.w
        current_weight_list[k] = i.w
    return mean_x, mean_y
```



S4 [t0]: Resample :based on sampling strategy, sample points from existing particles and their weights. Rule for this one : *Sample more from particle with higher weight*

: Target Object in Scene

: Initial Guesses of positions aka particles

: Particles after applying motion model

: Predicted particle for current time

```
new_index = np.random.choice(a=current_index,
                             size=N,
                             replace=True,
                             p=current_weight_list_norm)
```

Go to S1[t1] -> S2[t1] -> S3[t1] -> S4[t1] and keep on iterating…..