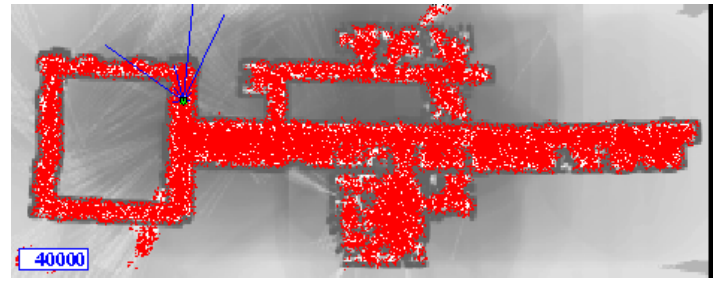
Particle filter

Nacho Sañudo
University of Modena and Reggio Emilia
Ignacio.sanudoolmedo@unimore.it

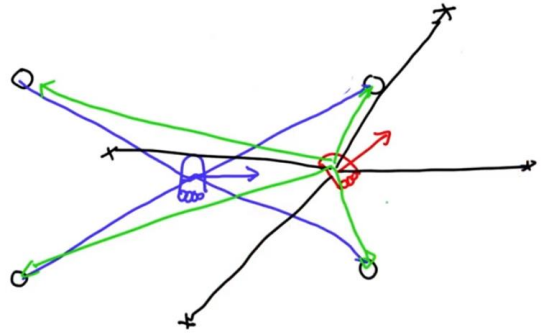
Credits to Udacity self-driving course

Particle filter

- **Objective**: find out the vehicles' location
  - Suppose that we are using a radar/LiDAR sensor that is able to extract the features of the map
- The particle filters represents the possible location using particles (red dots)
- A particle (a single guess) is composed of **x,y and heading (θ)** direction components
- **At the beginning the particles are uniformly spread**
  - **the particle filter makes survive the particles in proportion**
    - **how consistent the guess of the position is**

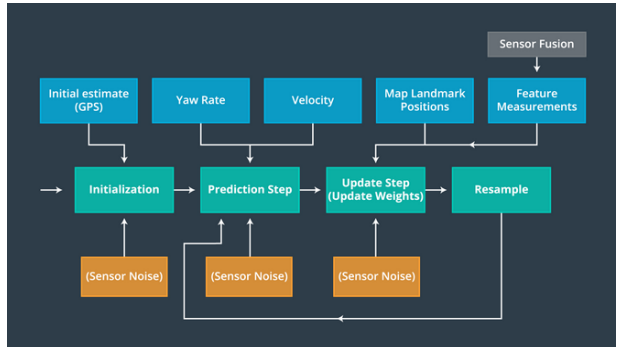


Particle filter



- Particles survive considering how consistent they are with respect the vehicle sensor measurements
  - The closer the particle is to the observed measurements the more likely the position will be correct
- **The difference of the *projected measurements (black)* and *predicted measurement (green)* leads to compute the weight**
  - The larger the weight the more important the particle is
- Through the time some particle survive and some not
  - The probability to survive will be proportional to its weight (**importance weight**)
  - The **resampling** technique allows us to randomly replace low weight particles

Algorithm



Algorithm Particle_filter($\mathcal{X}_{t-1}, u_t, z_t$):

$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$ → Initialize

for $m = 1$ *to* M *do*

  sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$ → Prediction

  $w_t^{[m]} = p(z_t \mid x_t^{[m]})$

  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ → Update

endfor

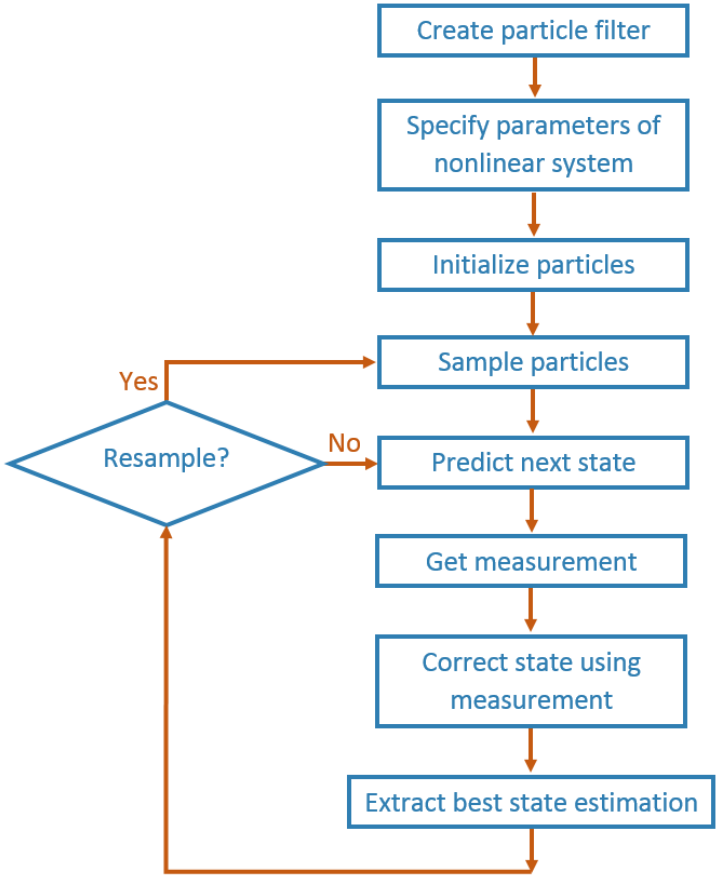
for $m = 1$ *to* M *do*

  draw i *with probability* $\propto w_t^{[i]}$

  add $x_t^{[i]}$ *to* $\mathcal{X}_t$ → Resampling

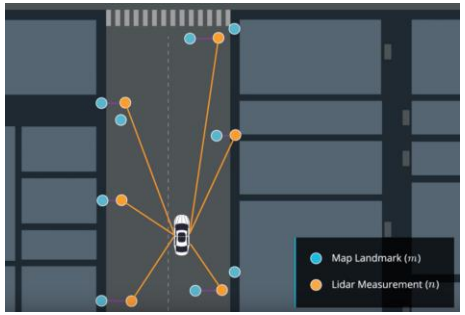
endfor

return $\mathcal{X}_t$

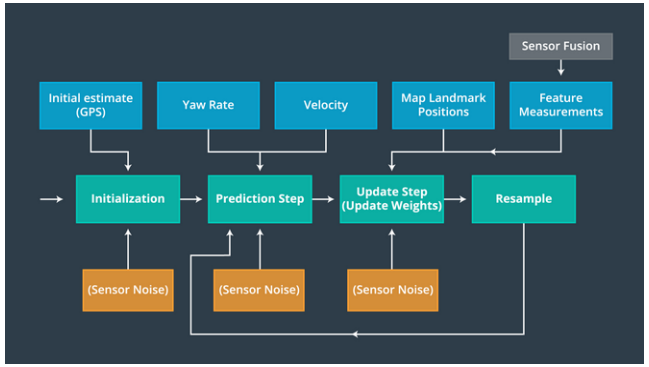


Input algorithm

- The map is given
  - Coordinates [*x,y*] of the landmarks in the map
- We use the sensors of the car to detect the landmarks
  - We assume that each time the sensors produce data, an algorithm as a black box returns the x,y coordinates of the landmarks with respect the position of the vehicle



Initialization



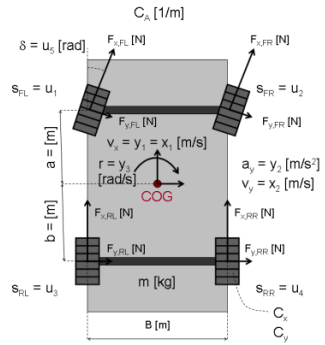
- Initiliaze all the particles
  1. **<u>Decide how many particles use (TODO)</u>**

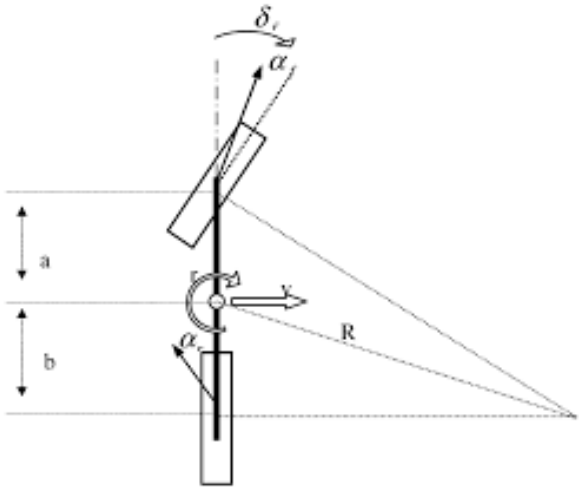
     Too many is very “expensive”

     Few particles do not allow to cover all the likely positions
  2. We use the GPS as an initial estimate
     - If not we can distribute uniformly the particles on the map
  3. Each particle is characterized by a gaussian that is influenced by the sensor noise

Prediction - Motion model

- **Where the vehicle will be at the next time interval?**
  - We need to define a motion model
- Motion model is a description of the physics of the vehicle
- We will consider the 2D bicycle model
  - The front wheels are connected to the rear wheels and we assume that the two front wheels (and rear) are connected



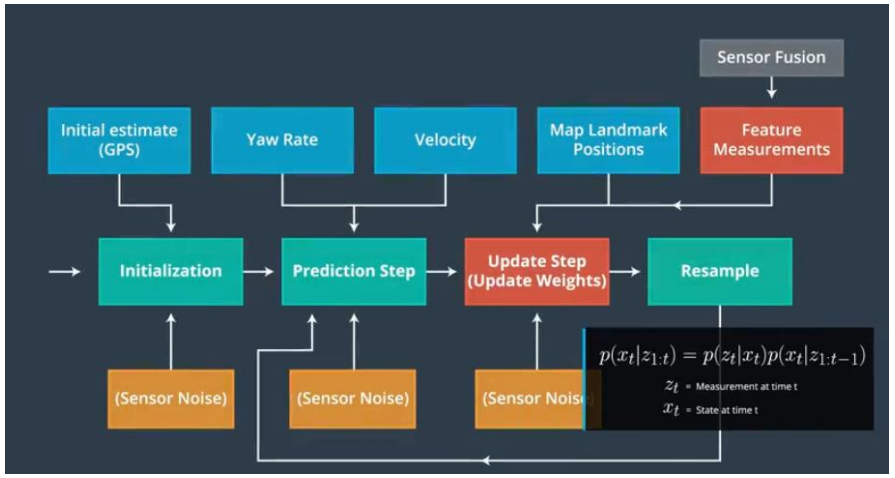


Particle filter - predict

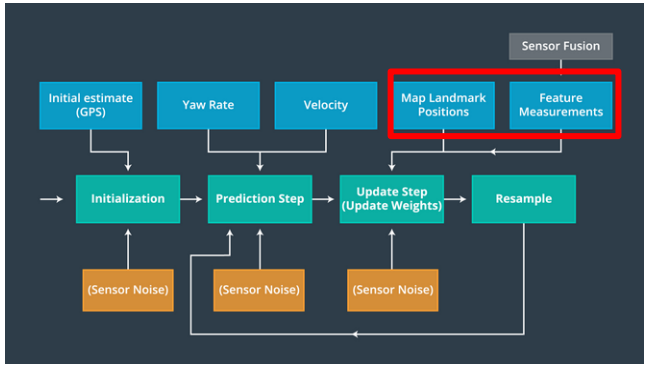
- **Where the vehicle will be at the next time interval?**
- For each particle we have to change (**predict**) the particles location based on velocity, yaw rate (adding gaussian noise) (**TODO**)
- *If the yaw rate (θ) is 0*
  - $x_f = x_0 + v(dt)(cos(\theta_p))$
  - $y_f = y_0 + v(dt)(sin(\theta_p))$
  - $\theta_f = \theta_0$
    - *x_f (final position) | x_0 (initial position) | dt (time elapsed) | θ (yaw rate)*
- *Otherwise*
  - $x_f = x_0 + \frac{v}{\dot{\theta}}[\sin\left(\theta_0 + \dot{\theta}(dt)\right) - (sin(\theta_0)]$
  - $y_f = y_0 + \frac{v}{\dot{\theta}}[\cos(\theta_0) - \cos(\theta_0 + \dot{\theta}(dt))]$
  - $\theta_f = \theta_0 + \dot{\theta}(dt)$
- In the code we will also add some random noise

Particle filter - Update

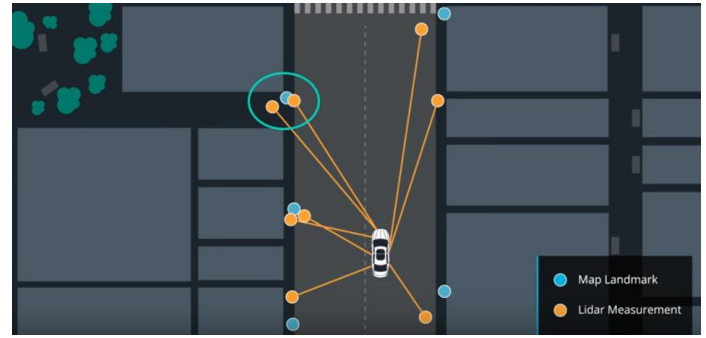
- **Pipeline:**
  1. Consider only the measurements that are in the range of the sensor
  2. Transform observations into map coordinates
  3. Associate observations with nearest landmarks
  4. Compute the probability of the particle



Update (Data Association)

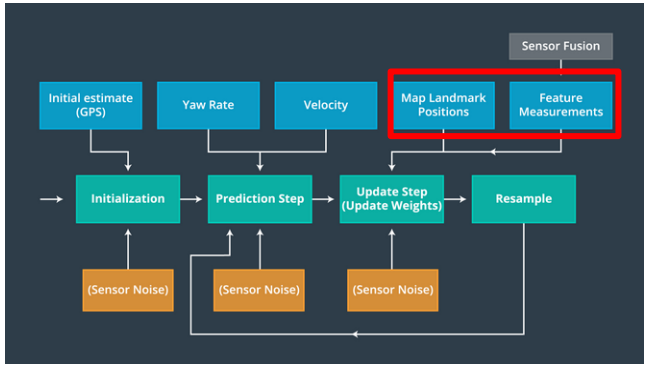


- The car might measure one landmark with multiple measurements
  - **Before we use landmark measurements to update the belief of our position, we have to solve the data association problem, i.e, match landmark measurements to objects in the real world (like map landmarks)**
- We use the **nearest neighbor** technique, but it has disadvantages, for instance,
  - - O(mn) where m is the number of landmarks and n is the number of sensor measurements…
  - - If a sensor is very noisy you could easily match spurious feature
  - …..



PROS	CONS
• Easy to understand • Easy to implement • Works well in many situations	• Not robust to high density of measurements or map landmarks • Not robust to sensor noise • Not robust to errors in position estimates • Inefficient to calculate • Does not take different sensor uncertainties into account

Update (Data Association)



For each observation in J //use the size() function

Min_dist=INFINITY

For each prediction in K

diff_x=predicted[K].x-observation[J].x

diff_y=predicted[K].y-observation[J].y

dist=sqrt(diff_x * diff_x + diff_y * diff_y);

if(dist<min_dist)

min_dist=dist

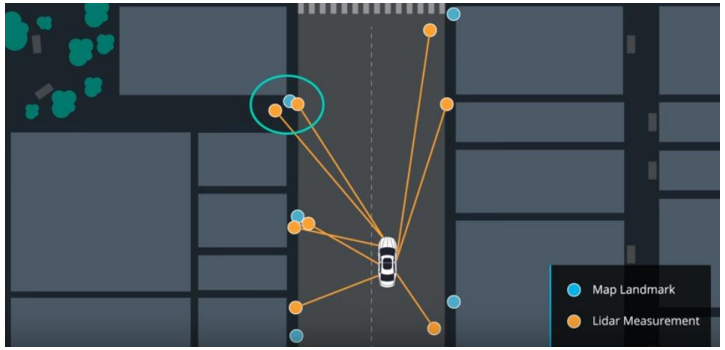
index_min=predicted[K].id

end if

End for

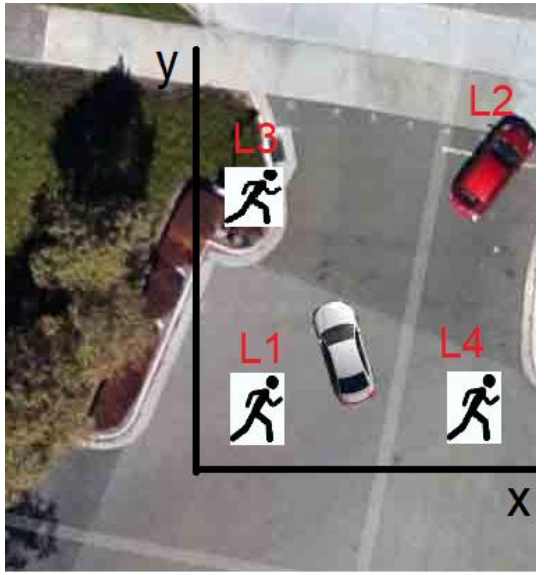
Observations[J].id=index_min

End for



Update (Coordinates transformation)

- Coordinates are given from car's perspective
- Homogenous transformation
  - Rotation of the maps' frame to match the particle's point of view



$$\begin{pmatrix} \cos\theta & -\sin\theta & x_t \\ \sin\theta & \cos\theta & y_t \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta + x_t \\ x\sin\theta + y\cos\theta + y_t \\ 1 \end{pmatrix}.$$

localMap.x = observation.x*cos(particle.theta)-observation.y*sin(particle.theta)+particle.x

.....

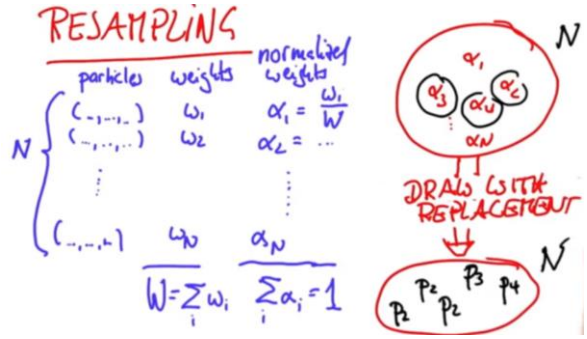
Update (Importance weight)

- We will use the **multivariate gaussian** to update the state of the weights of the particles, i.e., computing the probability of each particle as a candidate of the vehicles' position
  - how likely a set of landmarks measurements are, given a prediction state of the car and the assumption that the sensors has a sensor noise
    - Highly likely particles, i.e., particles that are close to the car's measurement might survive

$$f_{\mathbf{X}}(x_1, \ldots, x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k \mid \boldsymbol{\Sigma}\mid}}$$

$$x_i = measurement$$
$$\mu = predicted\ measurement$$
$$\Sigma = covariance\ of\ measurements$$

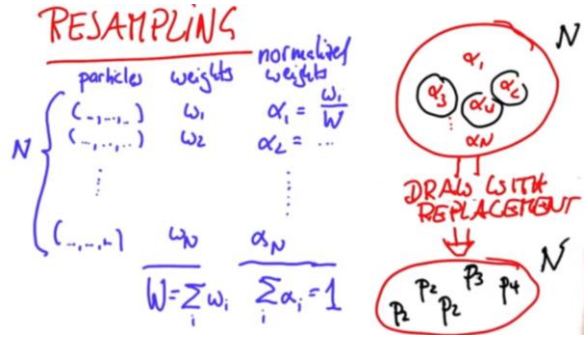
Particle filter (resampling)



- Sample particles from p with the probability that is proportional to its corresponding weight value
  - The larger the weight the higher the probability to survive
- Steps
  - Sample N particles with the weights
  - Normalized the weights
  - Replace low weight particles
    - replacing new particles from the old ones with replacement in proportion to the importance weight

N=6			
	P1	W1 = 1.4	P(P1) = ?
	P2	W2 = 0.5	P(P2) = ?
	P3	W3 = 0.9	P(P3) = ?
	P4	W4 = 3.2	P(P4) = ?
	P5	W5 = 2.1	P(P5) = ?
	P6	W6 = 0.1	P(P6) = ?

Particle filter (resampling)



- Sample particles from p with the probability that is proportional to its corresponding weight value
  - The larger the weight the higher the probability to survive
- Steps
  - Sample N particles with the weights
  - Normalized the weights
  - Replace low weight particles
    - replacing new particles from the old ones with replacement in proportion to the importance weight

N=6

P1	W1 = 1.4	$P(P1) = \alpha_1 = 1.4/8 = 0.175$
P2	W2 = 0.5	$P(P2) = \alpha_2 = 0.5/8 = 0.0625$
P3	W3 = 0.9	$P(P3) = \alpha_3 = 0.9/8 = 0.1125$
P4	W4 = 3.2	$P(P4) = \alpha_4 = 3.2/8 = 0.4$
P5	W5 = 2.1	$P(P5) = \alpha_5 = 2.1/8 = 0.2625$
P6	W6 = 0.1	$P(P6) = \alpha_6 = 0.1/8 = 0.0125$

What is the probability of never sampling P4? And P1?

$$\Sigma w_i = 1.4+0.5+0.9+3.2+1.9+0.1 = 8$$

Particle filter (resampling)

N=6

P1	W1 = 1.4	P(P1) = α_1 = 1.4/8 = 0.175
P2	W2 = 0.5	P(P2) = α_2 = 0.5/8 = 0.0625
P3	W3 = 0.9	P(P3) = α_3 = 0.9/8 = 0.1125
P4	W4 = 3.2	P(P4) = α_4 = 3.2/8 = 0.4
P5	W5 = 2.1	P(P5) = α_5 = 2.1/8 = 0.2625
P6	W6 = 0.1	P(P6) = α_6 = 0.1/8 = 0.0125

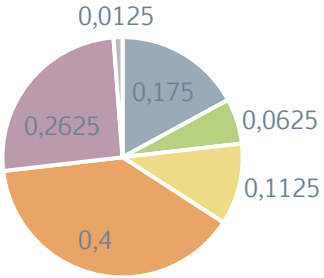
$$\Sigma w_i = 1.4+0.5+0.9+3.2+1.9+0.1 = 8$$

What is the probability of never sampling P4? And P1?

$$\Sigma w_i(!\,p4) = 0.6^5 = 0.077$$

$$\Sigma w_i(!\,p1) = 0.825^5 = 0.382$$

Particle filter (resampling wheel)

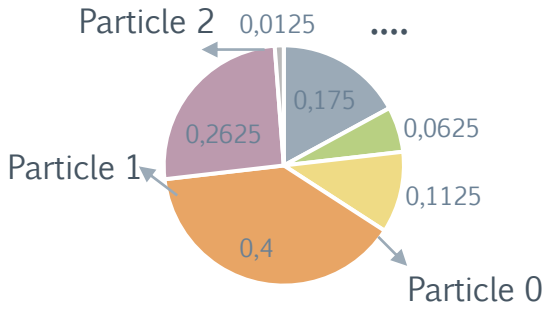


- **We would like to create a new list of particles with the sampling probability proportional to the importance weight**
- Resampling wheel is one common technique to implement resampling
  - Each particle occupies an slice of the wheel
  - Weights are proportional to the slices on the wheel
  - We choose any index from 0 to number of particles -1
  - We initialize a β value as 0
  - We iterate through all the particles to pick the particles
- At the end each particle is picked in proportion to the total circumference

```
particles = []
index = int(random.random()*NumParticles)
beta = 0.0
maxWeight = max(weight)

for i in range(NumParticles):
        beta = beta+random.random()*2* maxWeight
        while weight[index] < beta:
                beta = beta - weight[index]
                index = (index + 1) % NumParticles
        particles.append(p[index])
```

Particle filter (resampling wheel)

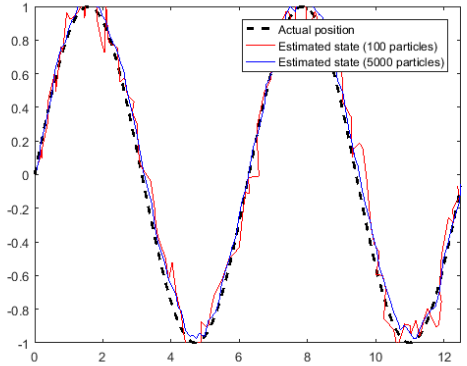


1. Pick a random index in the slice
2. For each particle we add to the variable “beta” a random value between 0 and 2*mw
3. While we don’t find a weight that is smaller than beta. We iterate and substract the weight of the current particle to beta and move the index to the next particle
4. If it is bigger we then take that particle

```
particles = []
index = int(random.random()*NumParticles)
beta = 0.0
maxWeight = max(weight)

for i in range(NumParticles):
        beta = beta+random.random()*2* maxWeight
        while weight[index] < beta:
                beta = beta - weight[index]
                index = (index + 1) % NumParticles
        particles.append(p[index])
```

Particle filter - Error



- Two methods to compute the error of the localization

- $Error\ weighted = \frac{\sum_{i=1}^{Np} w_i \sqrt{|p_i - gt|}}{\sum_{i=1}^{N_P} w_i}$
- $Error\ best = \sqrt{|p_{\max\_weight} - gt|}$

$\boldsymbol{Np}$ – *number of particles*
$\boldsymbol{w_i}$ - *weight of particle i*
$\boldsymbol{gt}$ – *ground truth*
$\boldsymbol{p_{max\_weight}}$ - *particle with the maximum weight*

Particle filter (TODOs)

- **ParticleFilter::init**
  - Num_particles (?)
- **ParticleFilter::prediction**
  - Fill the motion prediction formula
- **ParticleFilter::resample**
  - Write the resample algorithm replacing the particles assuming the particles' weight
- **ParticleFilter::dataAssociation**
  - Find the predicted measurement that is closest to each observed measurement and assign the observed measurement to this particular landmark (using the nearest neighbor alg.)

KF vs PF vs UKF vs EKF

- While Kalman filter can be used for linear or linearized processes and measurement system, the particle filter can be used for nonlinear systems
- The uncertainty of Kalman filter is restricted to Gaussian distribution, while the particle filter can deal with non-Gaussian noise distribution
- **When sensor noise exhibits jerky error, Kalman filter results in location estimation with hopping while particle filter still produces robust localization**
- **In a nutshell, the particle filter is robust, provides good localization estimates and is easy to implement**

Comparison of Kalman filter and particle filter used for localization of an underwater vehicle. Nak Yong Ko; Tae Gyun Kim