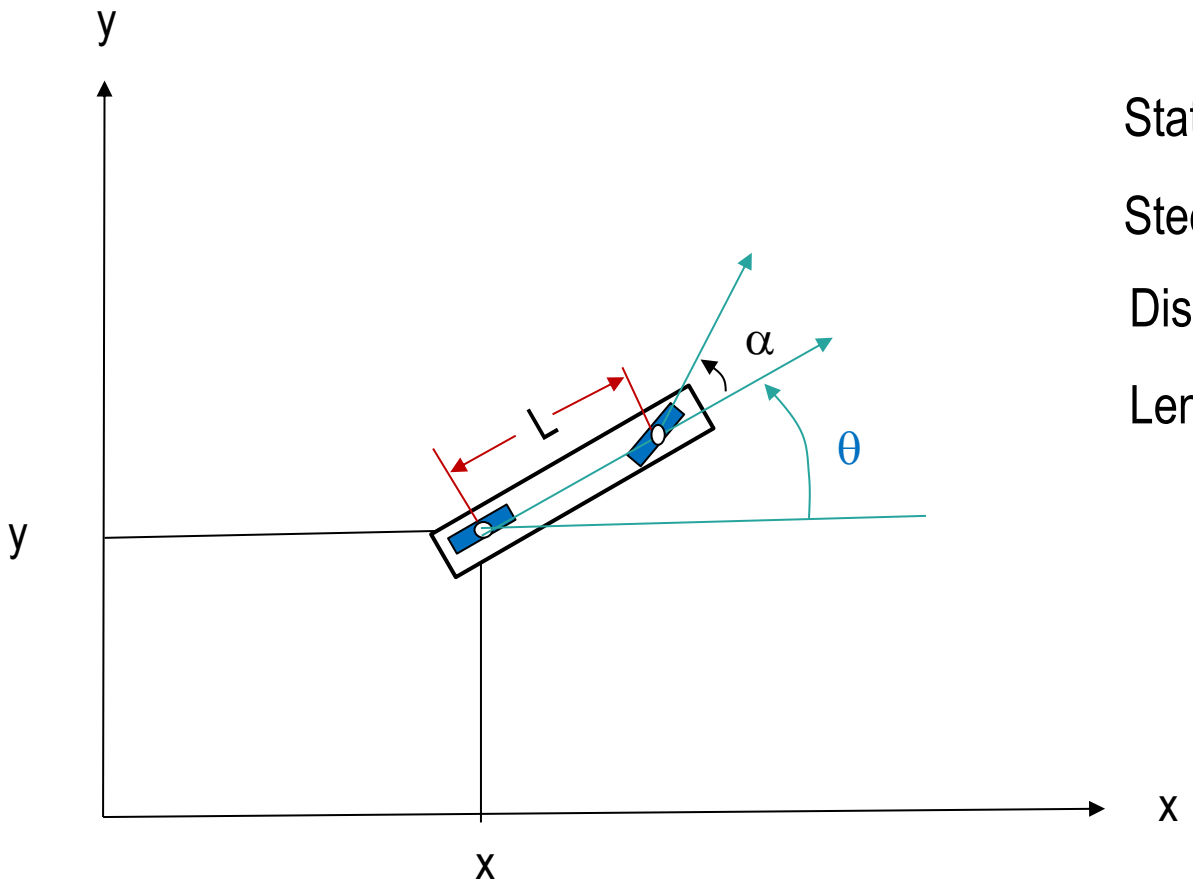
Mobile Robotics

Particle Filter Projects

Bicycle

- The bicycle robot shown below will be used in this project



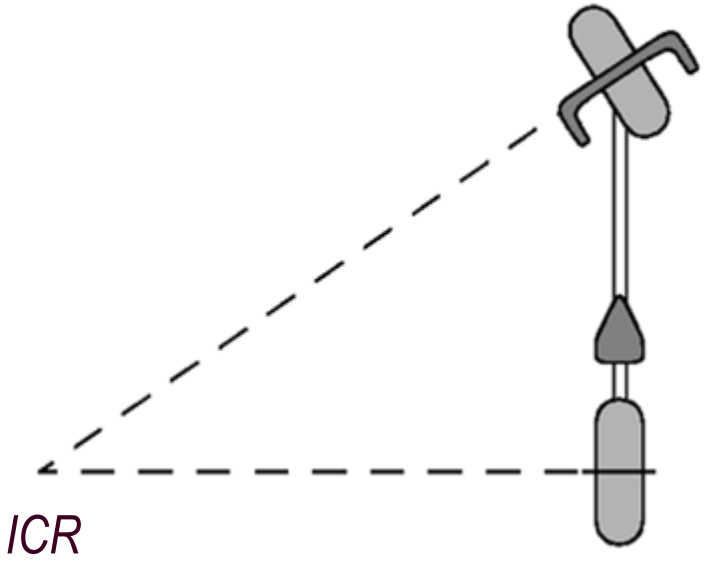
State: (x, y, θ)

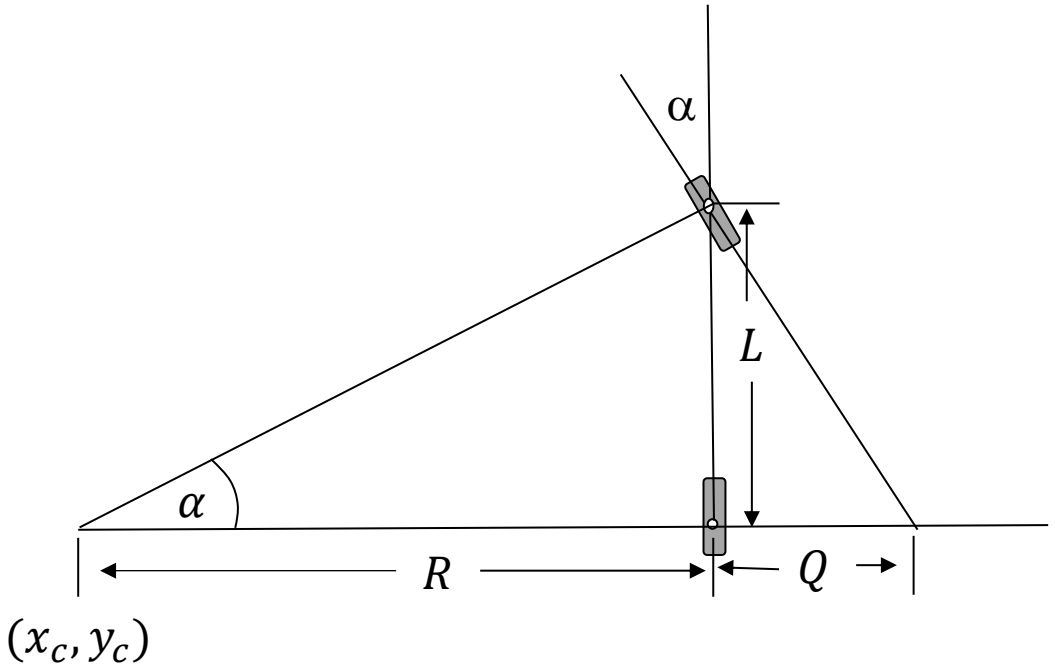
Steering angle : α

Distance: d

Length: L

Bicycle Maneuverability: Instantaneous Center of Rotation





$$\frac{Q}{L} = \frac{L}{R} = \tan(\alpha)$$

Bicycle Turning Angle β

State: (x, y, θ)

Steering angle : α

Distance: d

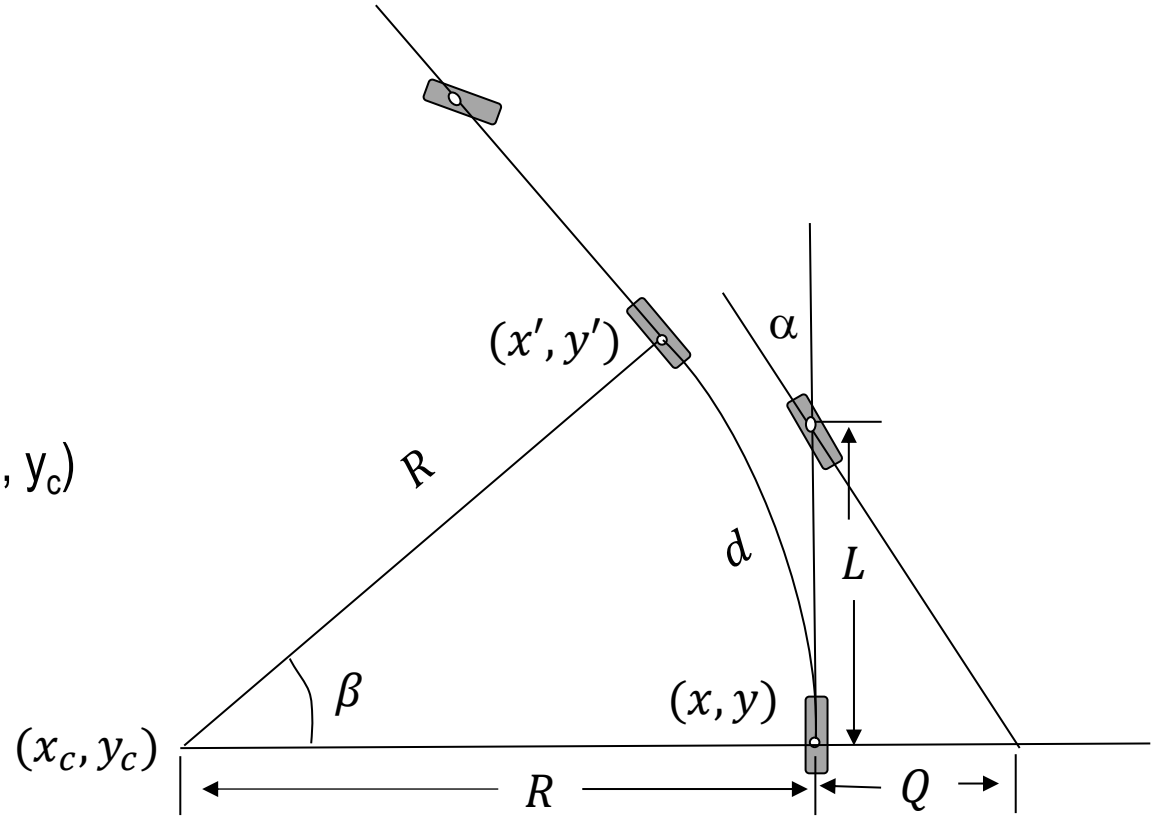
Length: L

New state: (x', y', θ')

Motion (Rotation) center: (x_c, y_c)

Motion (Rotation) radius: R

Rotation angle: β



$$R = {}^{d}/_{\beta}$$

$$\beta = \frac{d}{L} \cdot \tan\alpha$$

$$\frac{Q}{L} = \frac{L}{R} = \tan(\alpha)$$

Bicycle Motion Model (Kinematic Equations)

$$x_c = x - R \cdot \sin\theta$$

$$y_c = y + R \cdot \cos\theta$$

$$x' = x_c + R \cdot \sin(\theta + \beta)$$

$$y' = y_c - R \cdot \cos(\theta + \beta)$$

$$\theta' = (\theta + \beta) \bmod 2\pi$$

$|\beta| < 0.001$, use straight line

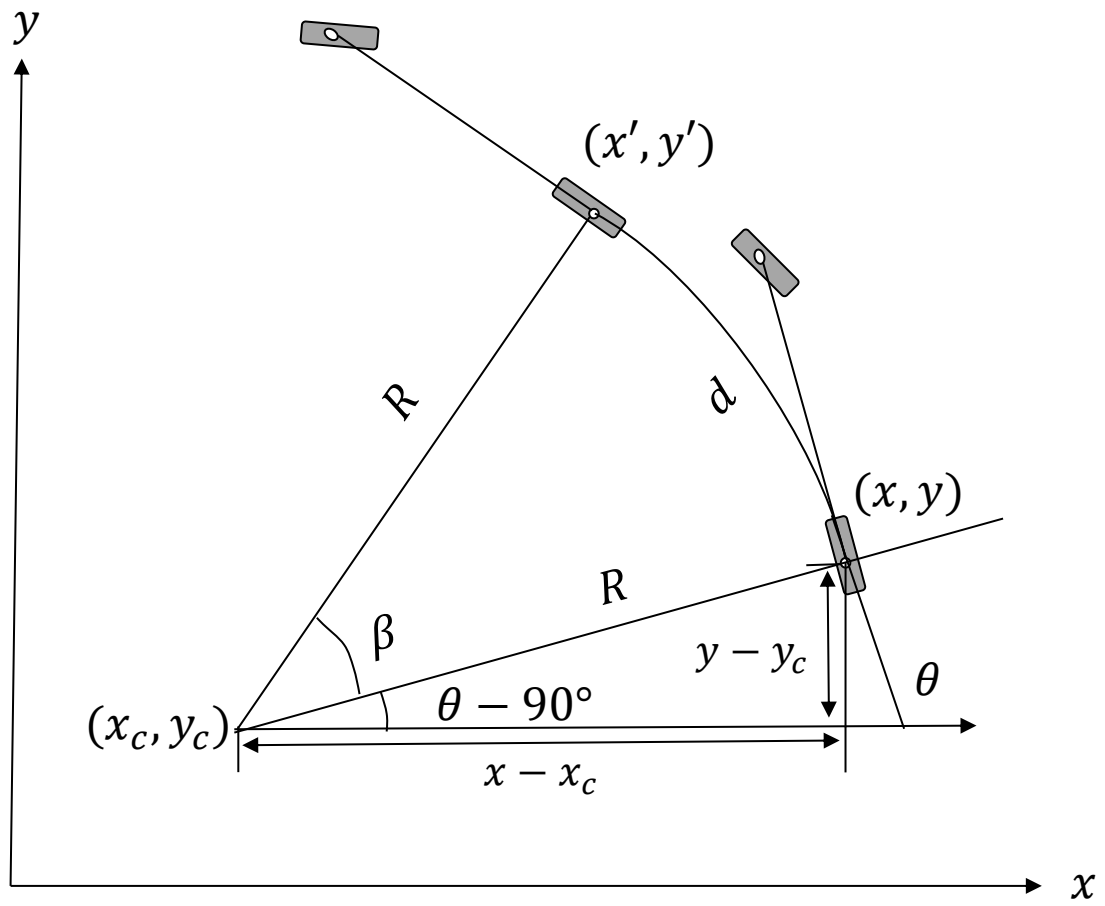
$$x' = x + d \cdot \cos(\theta)$$

$$y' = y - d \cdot \sin(\theta)$$

$$\theta' = (\theta + \beta) \bmod 2\pi$$

$$R = {}^{d}/_{\beta}$$

$$\beta = \frac{d}{L} \cdot \tan\alpha$$



$$x = x_c + R \cdot \sin\theta$$

$$y = y_c - R \cdot \cos\theta$$

Bicycle Motion Model (Kinematic Equations Cont.)

$$x_c = x - R \cdot \sin\theta$$

$$y_c = y + R \cdot \cos\theta$$

$$x' = x_c + R \cdot \sin(\theta + \beta)$$

$$y' = y_c - R \cdot \cos(\theta + \beta)$$

$$\theta' = (\theta + \beta) \bmod 2\pi$$

$$x' = x_c + R \cdot \sin(\theta + \beta)$$
$$= (x - R \cdot \sin\theta) + R \cdot \sin(\theta + \beta)$$
$$= x + R \cdot (\sin(\theta + \beta) - \sin\theta)$$
$$= x + d \cdot \frac{\sin(\theta + \beta) - \sin(\theta)}{\beta}$$

$|\beta| < 0.001$, use straight line

$$x' = x + d \cdot \cos(\theta)$$

$$y' = y - d \cdot \sin(\theta)$$

$$\theta' = (\theta + \beta) \bmod 2\pi$$

$when\ |\beta| < 0.001, (very\ small)$

$$\lim_{\beta \to 0} \frac{\sin(\theta + \beta) - \sin(\theta)}{\beta} = \sin'(\theta) = \cos(\theta)$$

$$x' = x + d \cdot \cos(\theta)$$

$$R = {}^{d}/_{\beta}$$

$$\beta = \frac{d}{L} \cdot \tan\alpha$$

Project (PP3A): Circular Move

PP3A-CircularMotion.py

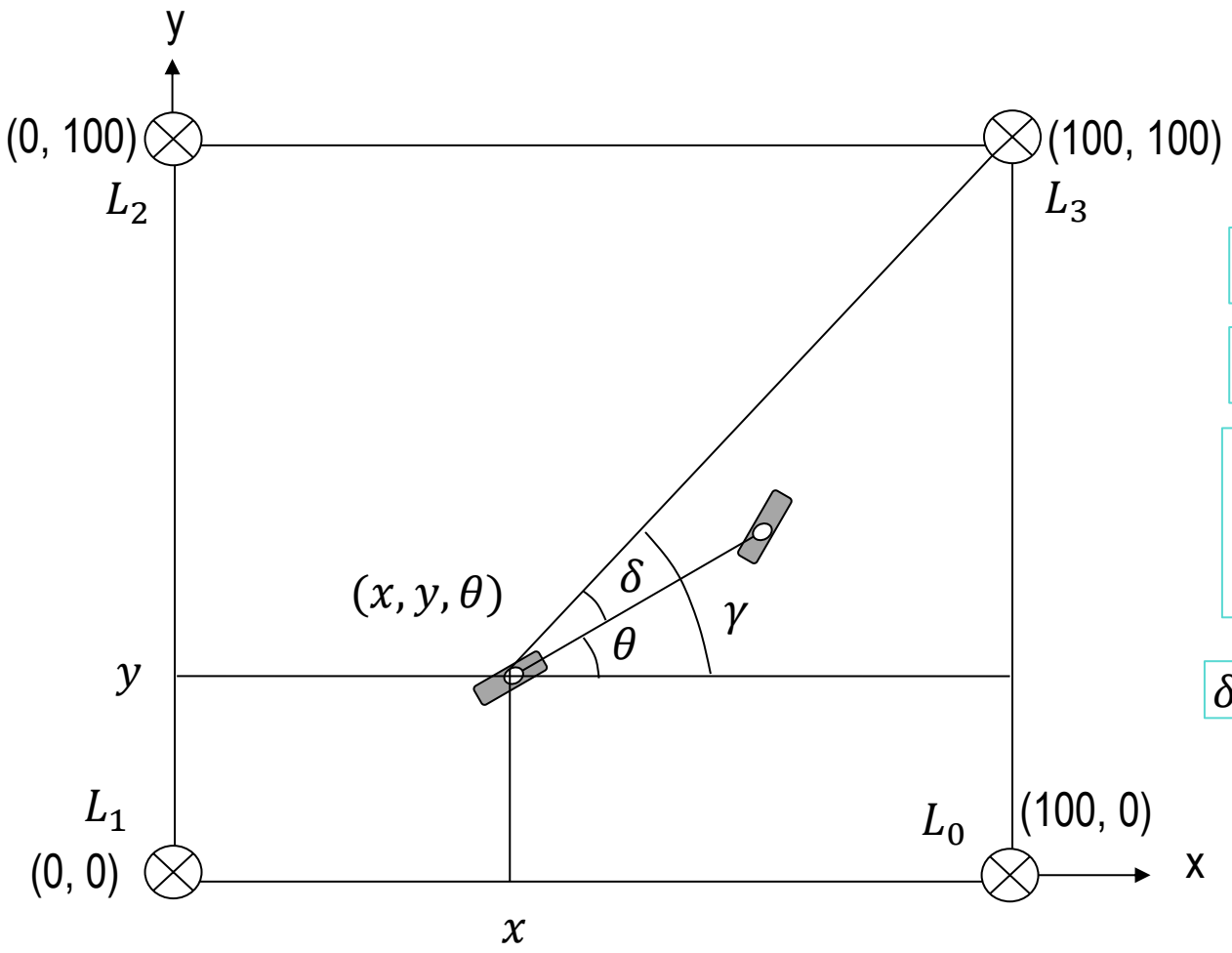
Write a function in the class robot - move(self, motion) which returns an instance of the robot class with the correct coordinates

```
##
myrobot = robot(length)
myrobot.set(0.0, 0.0, 0.0)
myrobot.set_noise(bearing_noise, steering_noise, distance_noise)
##
motions = [[0.0, 10.0], [pi / 6.0, 10], [0.0, 20.0]]
##
T = len(motions)
##
print ('Robot:    ', myrobot)
for t in range(T):
    myrobot = myrobot.move(motions[t])
    print ('Robot:    ', myrobot)
##
```

```
Robot:      [x=0.0 y=0.0 orient=0.0]
Robot:      [x=10.0 y=0.0 orient=0.0]
Robot:      [x=19.861 y=1.4333 orient=0.2886]
Robot:      [x=39.034 y=7.1270 orient=0.2886]
```

Robot World: Bearing Angles

- One robot and four landmarks inside a square are shown below.



Bearing angle: δ

Orientation: θ

Angle between line of robot to landmark and x-axis: γ

$$\delta = (\gamma - \theta)\ mod\ 2\pi$$

Project (PP3B): Sense

PP3B-Sensing.py

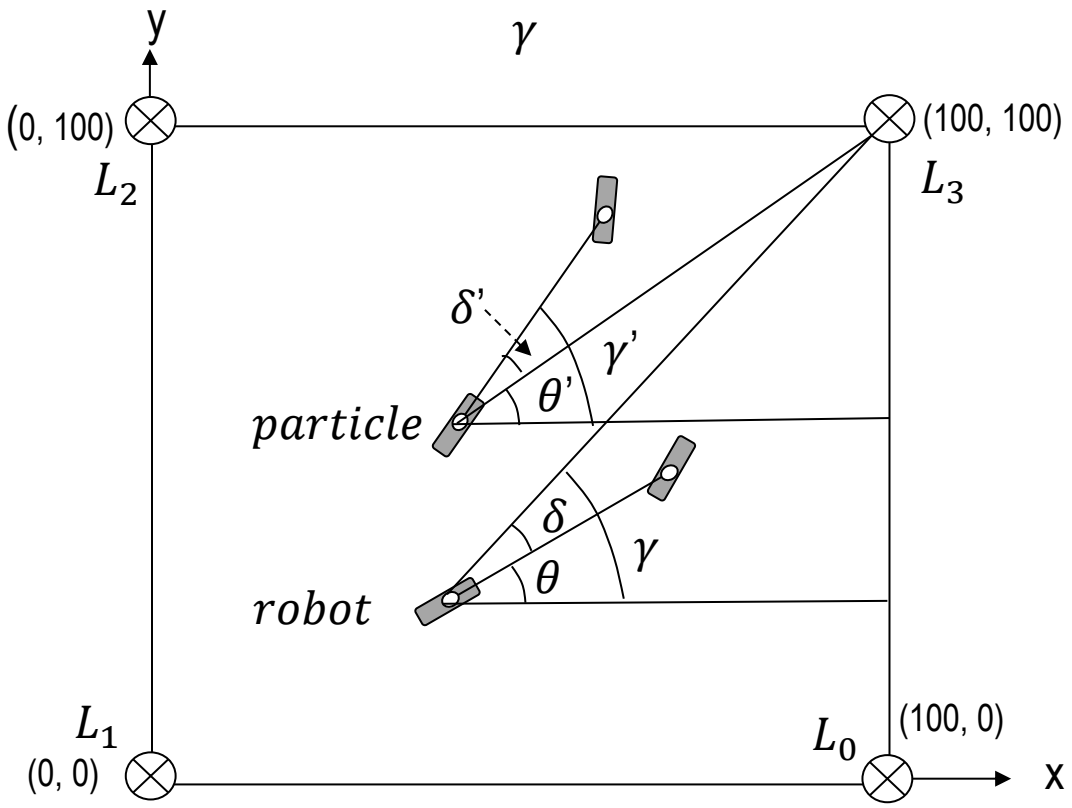
Write a function in the class robot - sense(self) which returns a list of four bearing angles to each of four different landmarks

```
##
length = 20.
bearing_noise  = 0.0
steering_noise = 0.0
distance_noise = 0.0
##
myrobot = robot(length)
myrobot.set(30.0, 20.0, 0.0)
myrobot.set_noise(bearing_noise, steering_noise, distance_noise)
##
print ('Measurements: ', myrobot.sense())
##
```

```
Measurements:  [6.004885648174475, 3.7295952571373605, 1.9295669970654687,
0.8519663271732721]
```

Importance Weight

- The closer the measurement of a particle is to the measurement of the robot, the more important the particle is.



Bearing angle: δ

Orientation:θ

Angle between line of robot to landmark and x-axis: γ

$$\delta = (\gamma - \theta)\ mod\ 2\pi$$

$$\delta' = (\gamma' - \theta')\ mod\ 2\pi$$

$$\sum |\delta' - \delta|$$

Importance Weight: Code

```
def measurement_prob(self, measurements):

    # calculate the correct measurement
    predicted_measurements = self.sense(0) # argument 0 will switch off noise.


    # compute errors
    error = 1.0
    for i in range(len(measurements)):
        error_bearing = abs(measurements[i] - predicted_measurements[i])
        error_bearing = (error_bearing + pi) % (2.0 * pi) - pi # truncate


        # update Gaussian
        error *= (exp(- (error_bearing ** 2) / (self.bearing_noise ** 2) / 2.0) /
                  sqrt(2.0 * pi * (self.bearing_noise ** 2)))

    return error
```

```
def sense(self, noisy = 1): #do not change the name of this function
```

Note: when argument noisy is 1, you should add noise in sense function, otherwise not.

Resampling Algorithm and Example

Residual resampling: a variation of roulette wheel sampling

```
p3 = []
index = int(random.random() * N)
beta = 0.0
mw = max(w)
for i in range(N):
    r = random.random()*2*mw
    beta += r
    while beta > w[index]:
        beta -= w[index]
        index = (index + 1) % N
    p3.append(p[index])
p = p3
```

N = 5
mw = 0.4
index = 2

i	0	1	2	3	4
w	0.10	0.20	0.40	0.10	0.20
r	0.25	0.40	0.20	0.25	0.15
beta	0.25	0.65-0.40 = 0.25 0.25-0.10 = 0.15	0.35-0.20 = 0.15 0.15-0.10 = 0.05	0.30-0.20 = 0.10	0.25
index	2	2->3->4	4->0->1	1->2	2
p3	x2	x4	x1	x2	x2

Resampling Algorithm and Example

Residual resampling

N = 10

mw = 0.25

index = 2

```
p3 = []
index = int(random.random() * N)
beta = 0.0
mw = max(w)
for i in range(N):
    r = random.random()*2*mw
    beta += r
    while beta > w[index]:
        beta -= w[index]
        index = (index + 1) % N
    p3.append(p[index])
p = p3
```

i	0	1	2	3	4	5	6	7	8	9
w	0.10	0.15	0.05	0.25	0.15	0.07	0.06	0.04	0.10	0.05
r	0.22	0.03	0.07	0.24	0.10	0.17	0.15	0.04	0.11	0.21
beta	0.22-0.05=0.17	0.20	0.27-0.25=0.02	0.26-0.15-0.07=0.04	0.14-0.06-0.04=0.04	0.21-0.10-0.05=0.06	0.21-0.10=0.11	0.15-0.15=0.00	0.11-0.05=0.06	0.27-0.25=0.02
index	2/3	3	3/4	4/5/6	6/7/8	8/9/0	0/1	1/2	2/3	3/4
s	x3	x3	x4	x6	x8	x0	x1	x2	x3	x4

Project (PP3C): Putting Everything together

PP3C-PutItTother.py

- First make sure that your sense and move functions work as expected for the test cases provided at the bottom of the previous two programming projects
- Copy your sense and move definitions into the robot class, BUT now include noise
  - A good way to include noise in the sense step is to add Gaussian noise, centered at zero with variance self.bearing_noise to each bearing. You can do this with the command random.gauss(0, self.bearing_noise)
  - In the move step, you should make sure that your actual steering angle is chosen from a Gaussian distribution of steering angles. This distribution should be centered at the intended steering angle with variance of self.steering_noise.
- A particle filter code is given. The particle filter will call your sense and move functions.

Generate Ground Truth

- The following function generates a sequence of measurements as an assumed true robot moves.
  - These measurements will be used to compute the importance weight of a particle.

```
def generate_ground_truth(motions):

    myrobot = robot()
    myrobot.set_noise(bearing_noise, steering_noise, distance_noise)

    Z = []
    T = len(motions)

    for t in range(T):
        myrobot = myrobot.move(motions[t])
        Z.append(myrobot.sense())
    print ('Robot:    ', myrobot)
    return [myrobot, Z]
```

Extract Position From a Particle Set

- The following function computes the center of a set of particles.
  - The center of particles will be used as an estimated position of robot.

```
def get_position(p):
    x = 0.0
    y = 0.0
    orientation = 0.0
    for i in range(len(p)):
        x += p[i].x
        y += p[i].y
        # orientation is tricky because it is cyclic. By normalizing
        # around the first particle we are somewhat more robust to
        # the 0=2pi problem
        orientation += (((p[i].orientation - p[0].orientation + pi) % (2.0 * pi))
                        + p[0].orientation - pi)
    return [x / len(p), y / len(p), orientation / len(p)]
```

Check Output

- The following code checks to see if your particle filter localizes the robot to within the desired tolerances of the true position.
- You can test whether your particle filter works using this function.
  - Note: Even for a well-implemented particle filter this function occasionally returns False. This is because a particle filter is a randomized algorithm.
  - Please make sure this function returns True at least 80% of the time.

```
def check_output(final_robot, estimated_position):

    error_x = abs(final_robot.x - estimated_position[0])
    error_y = abs(final_robot.y - estimated_position[1])
    error_orientation = abs(final_robot.orientation - estimated_position[2])
    error_orientation = (error_orientation + pi) % (2.0 * pi) - pi
    correct = error_x < tolerance_xy and error_y < tolerance_xy \
              and error_orientation < tolerance_orientation
    return correct
```

```
Ground truth:     [x=34.561 y=-9.456 orient=5.3212]
Particle filter:  [32.30222790047895, -9.19340580211005, 5.338875218561909]
Code check:       True
```