Lesson 3: Particle Filters

Three Types of Filters

Particle Filters are a sequence of algorithms for estimating the state of a system. Of the filters we cover in this class, particle filters are both the easiest to program and the most flexible.

Question 1 (State Space):

Which type of state space does each filter use? Continuous or discrete?

Check the appropriate box:

Answer:

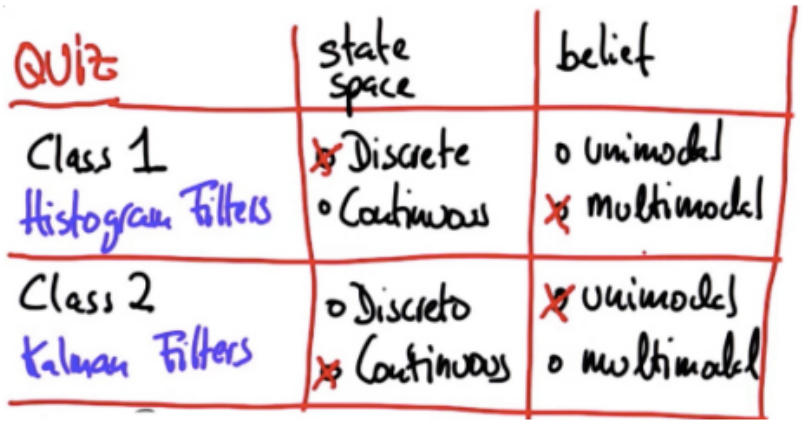
Histogram Filters have a discrete state space, while Kalman Filters have a continuous state space.

Question 2 (Belief Modality):

With respect to a belief function, check which distribution — unimodal or multimodal — pertains to each filter?

Answer: Belief Modality

Remember that even though the histogram filters are discrete, they are able to represent multiple bumps. On the other hand the Kalman filter was a single Gaussian, which is by definition unimodal.



Question 3 (Efficiency):

When it comes to scaling and the number of dimensions of the state space, how are grid cells, and/or Gaussians represented for each filter, as a quadratic or an exponential?

Answer: Efficiency

The histogram's biggest disadvantage is that it scales exponentially. This is because any grid that is defined over k-dimensions will end up having exponentially many grid cells in the number of dimensions, which doesn't allow us to represent high dimensional grids very well. This is sufficient for 3-dimensional robot localization programs, but becomes less useful with higher dimensions. In contrast, the Kalman filter is quadratic. It is fully represented by a vector — the mean — and the covariance matrix, which is quadratic. This makes the Kalman filter a lot more efficient because it can operate on a state space with more dimensions.

Question 4 (Exact or Approximate):

Do histogram filters and Kalman filters respectively give approximate or exact solutions?

Answer: Exact or Approximate

Both histogram and Kalman filters are not exact, but are an approximation of the posterior distribution. Histogram filters are approximate because the world is not discrete; localization for example, is an approximate filter. Kalman filters are also approximate, they are only exact for linear systems, however, the world is non-linear.

Particle Filters

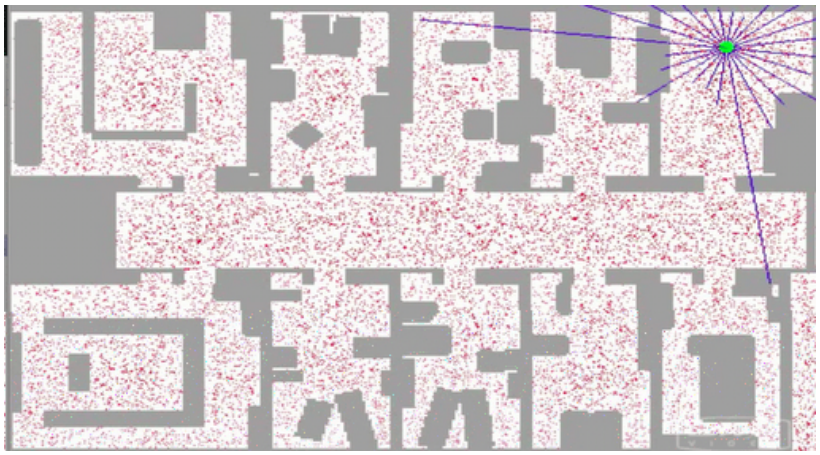
Let's fill in the characteristics of particle filters the same way we did with the histogram and Kalman filters.

Quiz	state space	belief	efficiency	in robotics
Class 1 Histogram Filters	x Discrete o Continuous	o unimodal x multimodal	o quadratic x exponential	o exact x approximate
Class 2 Kalman Filters	o Discrete x Continuous	x unimodal o multimodal	x quadratic o exponential	o exact x approximate
Class 3 Particle Filters	Continuous	multimodal	?	approximate

In terms of efficiency, the verdict is still out. In certain situations particle filters scale exponentially, and it would be a mistake to represent particle filters over anything more than four dimensions. However, in tracking domains they tend to scale much better.

The key advantage of particle filters is that they are really easy to program. In this class you will write your own particle filter for a continuous value localization problem.

Here is a floor plan of an environment where a robot is located and the robot has to perform global localization. Global localization is when an object has no idea where it is in space and has to find out where it is just based on sensory measurements.



The robot, which is located in the upper right hand corner of the environment, has range sensors that are represented by the blue stripes. The sensors use sonar sensors, which means sound, to range the distance of nearby obstacles. These sensors help the robot determine a good posterior distribution as to where it is. What the robot doesn't know is that it is starting in the middle of a corridor and completely uncertain as to where it is.

In this environment the red dots are particles. They are a discrete guess as to where the robot might be. These particles are structured as an x coordinate, a y coordinate and also a heading direction — three values to comprise a single guess. However, a single guess is not a filter, but rather it is the set of several thousands of guesses that together generate an approximate representation for the posterior of the robot.

The essence of particle filters is to have the particles guess where the robot might be moving, but also have them survive, a kind of "survival of the fittest," so that particles that are more consistent with the measurements, are more likely to survive. As a result, places of high probability will collect more particles, and therefore be more representative of the robot's posterior belief. The particle together, make up the approximate belief of the robot as it localizes itself.

Using Robot Class

Sebastian has written some code that will allow us to make a robot move along the x and y coordinates as well as in the heading direction. Take a minute to familiarize yourself with this code and then see how you can use it.

```
from math import *
import random

landmarks = [[20.0, 20.0],
                      [80.0, 80.0],
                      [20.0, 80.0],
                      [80.0, 20.0]]
world_size = 100.0

class robot:
    def __init__(self):
        self.x = random.random() * world_size
        self.y = random.random() * world_size
        self.orientation = random.random() * 2.0 * pi
        self.forward.noise = 0.0;
        self.turn_noise      = 0.0;
        self.sense_noise    = 0.0;
```

Call a function 'robot' and assign it to a function myrobot

```
myrobot = robot()
# the parameters are the x and y coordinate and the heading in radians
myrobot.set(10.0, 10.0, 0.0)
print myrobot
```

- *[x=10.0 y=10.0 heading=0.0]*

Next, make the robot move:

```
    myrobot = robot()
    # the parameters are the x and y coordinate and the heading in radians
    myrobot.set(10.0, 10.0, 0.0)
    print myrobot
    # this means the robot will move 10 meters forward and will not turn
    myrobot = myrobot.move(0.0, 10.0)
    print myrobot
```

- *[x=20.0 y=10.0 heading=0.0]*

Now, make the robot turn:

```
    myrobot = robot()
    # this is the x and y coordinate and the heading in radians
    myrobot.set(10.0, 10.0, 0.0)
    print myrobot
    # this will make the robot turn by pi/2 and 10 meters
    myrobot = myrobot.move(pi/2, 10.0)
    print myrobot
```

- *[x=10.0 y=20.0 heading=1.5707]*

You can generate measurements with the command sense to give you the distance to the four landmarks:

```
myrobot = robot()
# this is the x and y coordinate and the heading in radians
myrobot.set(10.0, 10.0, 0.0)
print myrobot
# this will make the robot turn by pi/2 and 10 meters
myrobot = myrobot.move(pi/2, 10.0)
print myrobot
print myrobot.sense()
```

- *[x=20.0 y=10.0 heading=0.0] [x=10.0 y=20.0 heading=1.5707] [10.0, 92.195444572928878, 60.87625302982199, 70.0]*

Robot Class Details

Included in the code that Sebastian has provided are a few functions to take note of:

```
class robot:
     def __init__(self):
         self.x = random.random() * world_size
         self.y = random.random() * world_size
         self.orientation = random.random() * 2.0 * pi
         self.forward.noise = 0.0;
         self.turn_noise    = 0.0;
         self.sense_noise   = 0.0;
```

The section of code above shows how the robot assimilates noises, which at this point are all set to zero.

```
     def set(self, new_x, new_y, new_orientation):
          if new_x < 0 or new_x >= world_size:
              raise ValueError, 'X coordinate out of bound'
          if new_y < 0 or new_y >= world_size:
              raise ValueError, 'Y coordinate out of bound'
          if new_orientation < 0 or new_orientation >= 2 *pi:
              raise ValueError, 'Orientation must be in [0..2pi]'
          self.x = float(new_x)
          self.y = float(new_y)
          self.orientation = float(new_orientation)

      def set_noise(self, new_f_noise, new_t_noise, new_s_noise):
          # makes it possible to change the noise parameters
          # this is often useful in particle filters
          self.forward_noise = float(new_f_noise);
          self.turn_noise    = float(new_t_noise);
          self.sense_noise   = float(new_s_noise);
```

The set_noise function above allows you to set noises.

```
    def measurement_prob(self, measurement):
         # calculates how likely a measurement should be
         # which is an essential step
     prob = 1.0;
     for i in range(len(landmarks)):
         dist = sqrt((self.x - landmarks[i][0]) ** 2 + (self.y - landmarks[i][1]) ** 2)
         prob *= self.Gaussian(dist, self.sens_noise, measurement[i])
     return prob
```

The function above, measurement_prob, accepts a measurement and tells you how plausible it is. This is a key aspect to the "survival of the fittest" aspect of particle filters. However, we will not use this function until later.

Question 5 (Moving Robot):

Using your interpreter, make a robot that satisfies the following requirements:

```
    # starts at 30.0, 50.0, heading north (=pi/2)
    # turns clockwise by pi/2, moves 15 meters
    # senses
    # turns clockwise by pi/2, moves 10 meters
    # sense
```

After printing senses the first time around we get the following output:

- *[39.0, 46.0, 39.0, 46.0]*

After printing sense the second time around we get the following output:

- *[32.0, 53.1, 47.1, 40.3]*

Answer: Moving Robot

```
myrobot = robot()
myrobot.set(30.0, 50.0, pi/2)
myrobot = myrobot.move(-pi/2, 15.0)
print myrobot.sense()

myrobot = myrobot.move(-pi/2, 10.0)
print myrobot.sense()
```

- *[39.0, 46.0, 39.0, 46.0] [32.0, 53.1, 47.1, 40.3]*

Question 6 (Add Noise):

The following code has built in noise variables for forward, turn and sense:

```
   class robot:
        def __init__(self):
            self.x = random.random() * world_size
            self.y = random.random() * world_size
            self.orientation = random.random() * 2.0 * pi
            self.forward_noise = 0.0;
            self.turn_noise    = 0.0;
            self.sense_noise   = 0.0;
```

Further below in the code you can set the noises:

```
    def set_noise(self, new_f_noise, new_t_noise, new_s_noise):
        # makes it possible to change the noise parameters
        # this is often useful in particle filters
        self.forward_noise = float(new_f_noise);
        self.turn_noise = float(new_t_noise);
        self.sense_noise = float(new_s_noise);
```

In your code, set the values as follows:

```
    # forward_noise = 5.0, turn_noise = 0.1, sense_noise = 5.0
    # starts at 30.0, 50.0, heading north (=pi/2)
    # turns clockwise by pi/2, moves 15 meters
    # senses
    # turns clockwise by pi/2, moves 10 meters
    # sense
```

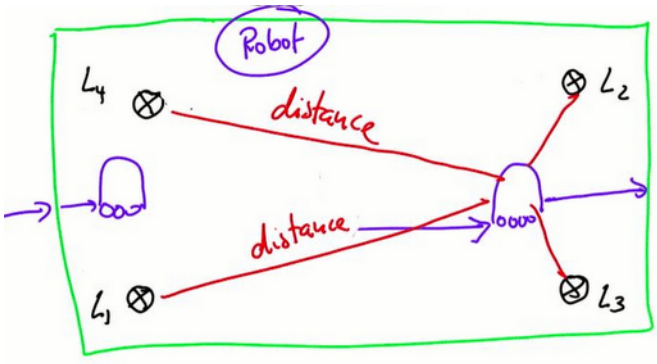
Answer: Add Noise

```
myrobot = robot()
myrobot.set_noise(5.0, 0.1, 5.0) # here is where you add your code
myrobot.set(30.0, 50.0, pi/2)
myrobot = myrobot.move(-pi/2, 15.0)
print myrobot.sense()

myrobot = myrobot.move(-pi/2, 10.0)
print myrobot.sense()
```

Notice that every time you hit run you get different set of values.

Robot World

Now, you can program the robot to turn, move straight after the turn, and sense the distance to four designated landmarks (L_1, L_2, L_3, L_4). The distances from the landmarks to the robot make up the measurement vector of the robot. The robot will live in a world of 100x100, so if it falls on one end, then it appears on the other — it is a cyclic world.



Creating Particles

The particle filter you are going to program maintains a set of 1,000 (N = 1000) random guesses as to where the robot might be, represented by a dot. Each dot is a vector that contains an x-coordinate (38.2), a y-coordinate (12.4), and heading direction (0.18). The heading direction is the angle (in radians) the robot points relative to the x-axis; so as this robot moves forward it will move slightly upwards.

In your code, every time you call the function robot() and assign it to a particle p[i], the elements p[i].x, p[i].y, p[i].orientation (which is the same as heading) are initialized at random.

In order to make a particle set of 1,000 particles, you have to program a separate piece of code that assigns 1,000 of those to a list.

Question 7 (Creating Particles):

Fill in the code so that your results assigns 1,000 particles to a list.

```
    N = 1000
    p = []
    #Your Code Here
    print len(p)
```

Answer: Creating Particles

```
N = 1000
p = []
for i in range(N): # iterate the loop 1000 times
    x = robot() # create an object called robot
    p.append(x) # append the object to growing list p
print len(p)
```

- *1000*

If you try to print just p you get 1000 particles, each of which has three values associated with it, an x-coordinate, a y-coordinate and an orientation (heading direction).

Question 8 (Robot Particles):

Take each particle and simulate robot motion. Each particle should first turn by 0.1 and then move forward 5.0 meters. Go back to your code and make a new set p that is the result of the specific motion, turn by 0.1 and move forward 5.0 meters, to all of the particles in p.

```
    N = 1000
    p = []
    for i in range(N):
        x = robot()
        p.append(x)
```

Answer: Robot Particles

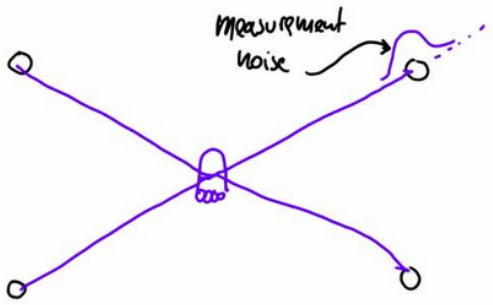
Here is one possible solution:

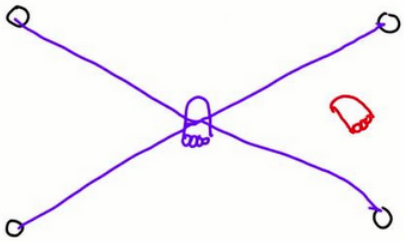
```
N = 1000 p = []
for i in range(N):
    x = robot()
    p.append(x)

p2 = []
for i in range(N): # go through all the particles again
    # append to list p2 the result of motion applied to the i particle,
    # chosen from particle set
    p2.append(p[i].move(0.1, 5.0))
p = p2
```

If you got this far, you are halfway through! However, the next half is going to be tricky.

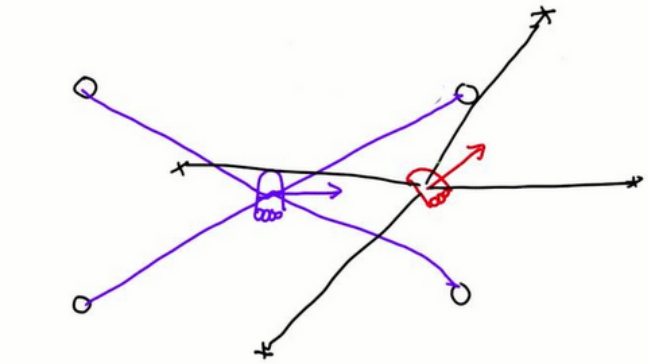
Second Half of Particle Filters

The second half of particle filters works like this; suppose you have a robot that sits amid four landmarks and can measure the exact distances to the landmarks. To the right, the image shows the robot's location and the distances it measures, as well as "measurement noise," which is modeled as a Gaussian with a mean of zero. This means there is a chance of the measurement being too short or too long, and that probability is governed by a Gaussian.



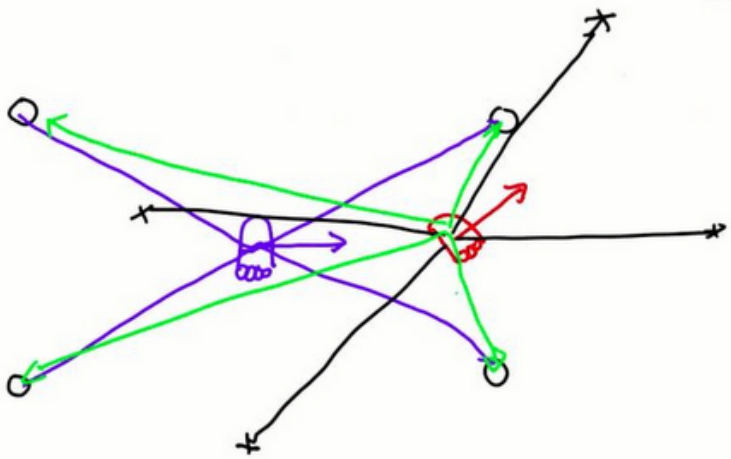


Now we have a measurement vector that consists of the four values of the four distances from L_1 to L_4. If a particle hypothesizes that its coordinates are somewhere other than where the robot actually is (the red robot indicates the particle hypothesis), we have the situation shown below.

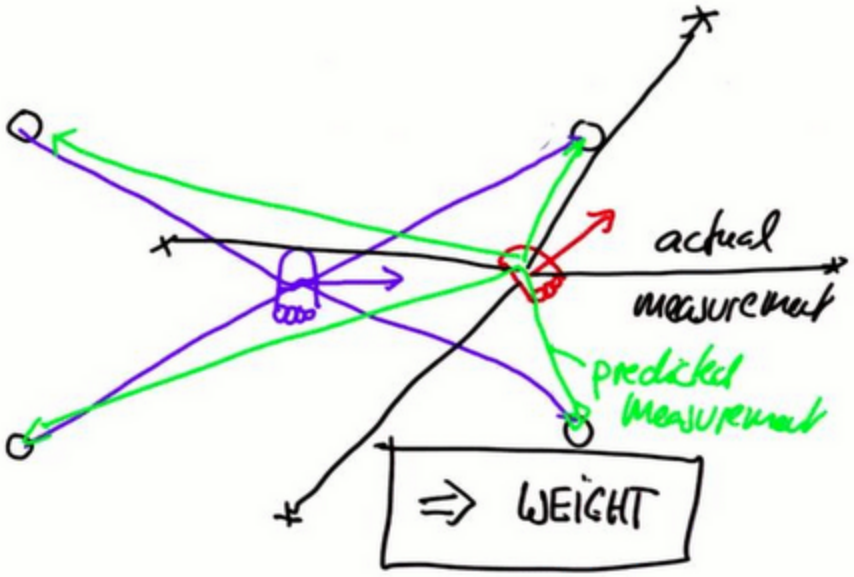


The particle also hypothesizes a different heading direction. You can take the measurement vector from our robot and apply it to the particle.

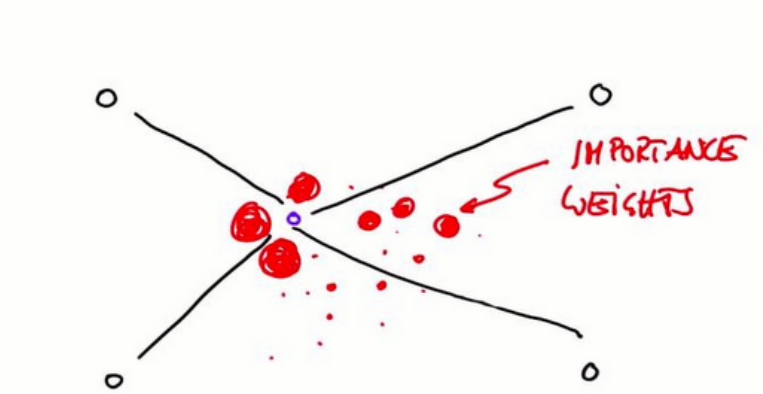
However, this ends up being a very poor measurement vector for the particle. The green indicates the measurement vector we would have predicted if the red particle actually were a good match for the robot's actual location.



The closer your particle is to the correct position, the more likely will be the set of measurements given that position. Here is the trick to particle filters; the mismatch of the actual measurement and the predicted measurement leads to an importance weight that tells you how important that specific particle is. The larger the weight the more important it is.



When you have a bunch of particles, each has its own weight; some are very plausible, while others look very implausible as indicated by the size of the particle.



Next we allow the particles to survive at random, but the probability of survival will be proportional to the weights. That is, a particle with a larger weight will survive at a higher proportion than a particle with a small weight. This means that after resampling, which is randomly drawing new particles from the old ones with replacement in proportion to the importance weight, the particles with a higher importance weight will live on, while the smaller ones will die out. The "with replacement" aspect of this selection method is important because it allows us to choose the high probability particles multiple times. This causes the particles to cluster around regions with high posterior probability.

From here you want to implement a method of setting importance weights, which is related to the likelihood of a measurement and you want to implement a method of resampling that grabs particles in proportion to those weights.

Have a look at this code:

```
# this is a random initialization of the robot, which will return a random outpu
myrobot = robot()
myrobot = myrobot.move(0.1, 5.0)
Z = myrobot.sense()
print Z
print myrobot
[69, 15, 53, 47]
[x=33.657 y=48.869 heading=0.5567]
```

Question 9 (Importance Weight):

Program a way to assign importance weights to each of the particles in the list. Make a list of 1,000 elements, where each element in the list contains a number that is proportional to how important the particle is. To make things easier Sebastian has written a function called measurement_probability. This function accepts a single parameter, the measurement vector Z that was just defined, and calculates as an output how likely the measurement is. By using a Gaussian, the function measures how far away the predicted measurements would be from the actual measurements.

```
def measurement_prob(self, measurement):
     # calculates how likely a measurement should be
     # which is an essential step
     prob = 1.0;
     for i in range(len(landmarks)):
         dist = sqrt((self.x - landmarks[i][0]) ** 2 + (self.y - landmarks[i][1]))
         prob *= self.Gaussian(dist, self.sense_noise, measurement[i])
     return prob
```

For this function to run properly, you have to assume that there is measurement noise for the particles, so we need to change our particle generator:

```
N = 1000
p = []
for i in range(N):
    x = robot()
    x.set_noise(0.05, 0.05, 5.0) # this line ensures particles have a certain amount of noi
    p.append(x)
```

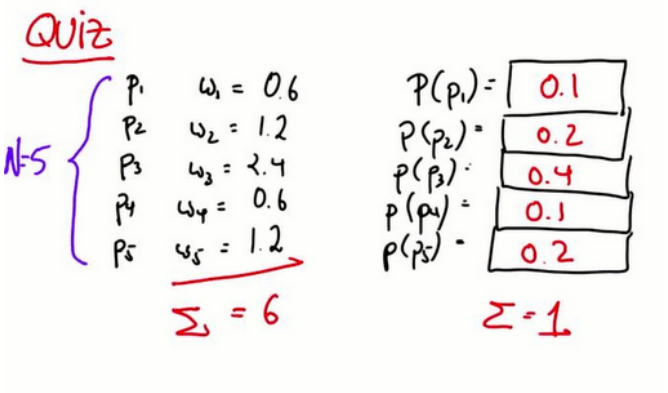
Once again, please program a list of 1,000 elements in w so that each number in this vector reflects the output of the function measurement_prob applied to the measurement Z. This will help us when we want to resample our particles to create a new set of particles that better match the position of our robot.

Answer: Importance Weight

```
w = []
for i in range(N):
    # append the output of the function measurement_prob
    # to the i particle with the argument of the extra measurement
    w.append(p[i].measurement_prob(Z))
print w
```

The results return some outputs that are highly unlikely, with exponents of -146, while others are more likely, for example exponents of -5 — these are the particles that are more likely to survive.

For the final step of the particle filter algorithm, you have to sample particles from p with a probability that is proportional to a corresponding w value. Particles in p that have a large output value should be drawn more frequently than the ones with a smaller value. How hard can that be?



Resampling

Resampling is the trickiest part of programming a particle filter. Resampling is when you generate a new list of particles by letting some of your old particles survive and killing off others. When you are given N particles to resample, each of them will have three values (x, y, and orientation) and a weight, w. The weights are continuous values which sum to W.

$$W = \sum_i w_i$$

We can normalize the weights:

$$\alpha_1 = \frac{w_1}{W}$$

$$\alpha_2 = \frac{w_2}{W}$$

$$\alpha_N = \frac{w_N}{W}.$$

The sum of all alphas (the normalized weights) is:

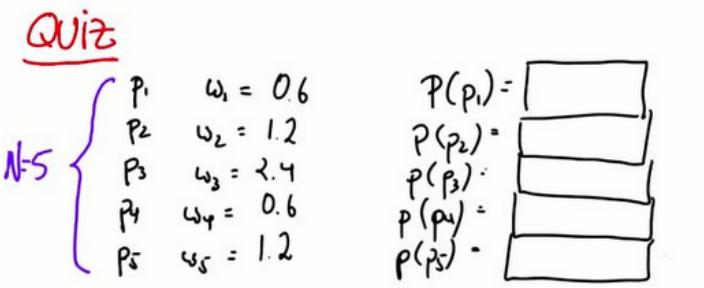
$$\sum_i \alpha_i = 1$$

Resampling puts all the particles and their normalized weights into a big bag. It then draws *N* new particles with replacement by picking each particle with a probability proportional to the value of it's α. For example, let's pretend we have 5 particles to be resampled with normalized weights of α_1 ,α_2 ,α_3,α_4, and α_5. The values of α_2 and α_3 are larger than the other 3; first we may draw α_2 , which becomes p_2, and similarly α_3 might also be large and picked up as p_3. By chance you may also pick up small α_4, to add p_4, and you can also pick the same one again, like α_2, to have two versions of p_2, or maybe even three!

If there are *N* particles to begin with, you draw *N* times. In the end, those particles that have a high normalized weight will occur more frequently in the new set. This is resampling.

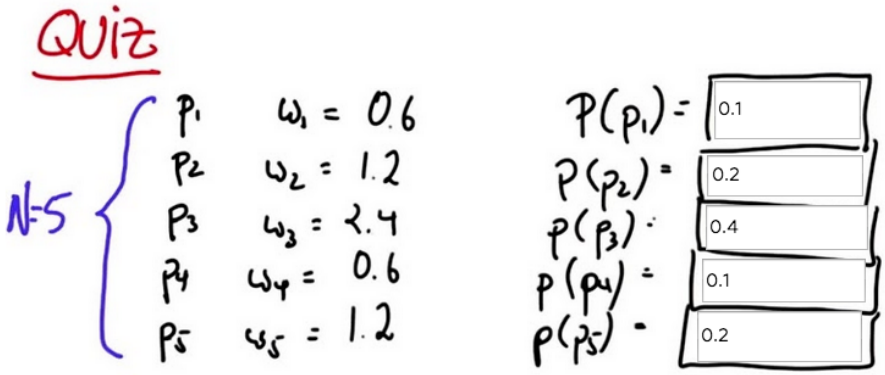
Question 10 (Resampling):

During the process of resampling, if you randomly draw a particle in accordance to the normalized importance weights, what is the probability of drawing p_1 - p_5?



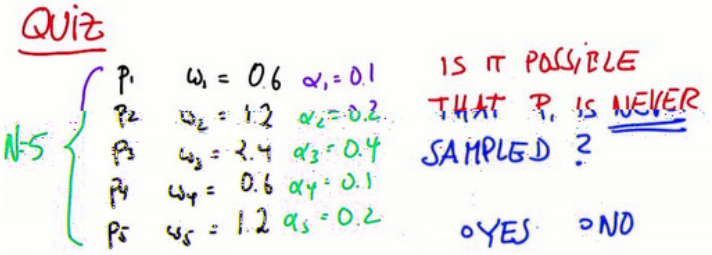
Answer: Resampling

To obtain the answer, you just have to normalize the importance weights by dividing each weight by the sum of the weights.



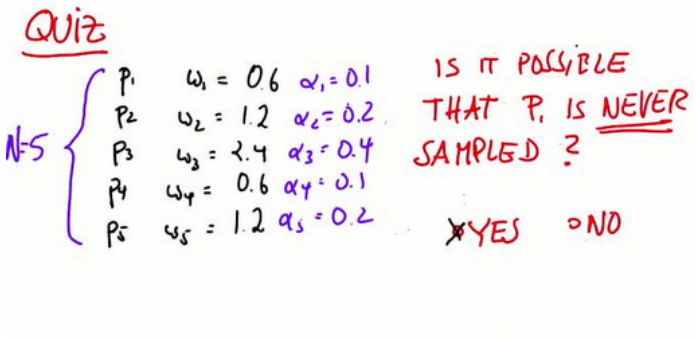
Question 11 (Never Sampled-1):

Is it possible that p_1 is never sampled in the resampling step? Check yes or no.



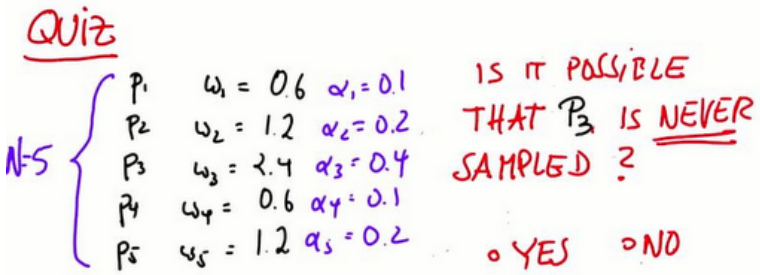
Answer: Never Sampled-1

Yes; something with an importance weight of 0.1 is quite unlikely to be sampled into the next data set.



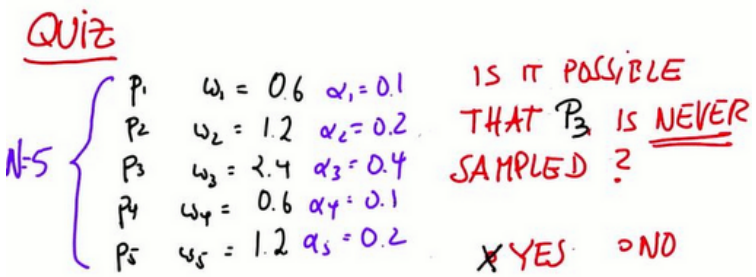
Question 12 (Never Sampled-2):

Is it possible that p_3 is never sampled in the resampling step? Check yes or no.



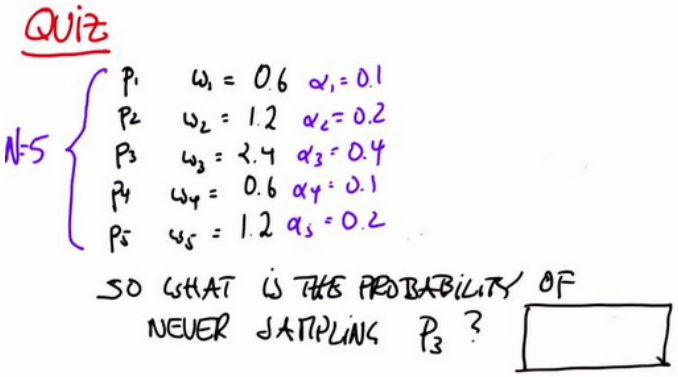
Answer: Never Sampled-2

The answer is yes again, because even though the importance weight is large, it is still possible that in each of the five resampling steps you would pick one of the other four.



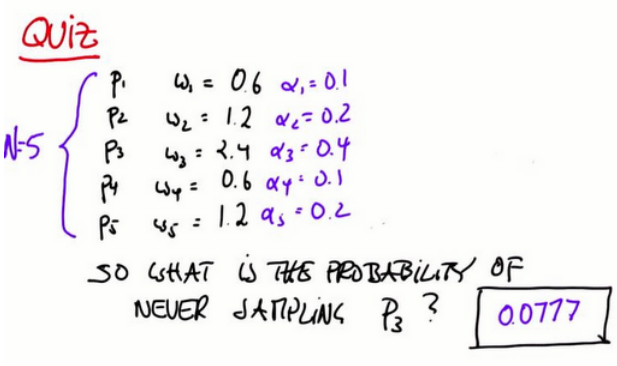
Question 13 (Never Sampled-3):

What is the probability of never sampling p_3? To answer this question, assume you make a new particle set with $N = 5$ new particles, where particles are drawn independently and with replacement.



Answer: Never Sampled-1

For p_3 to never be drawn in the resampling phase, you would always have to draw p_1, p_2, p_4 or p_5. These together have a 0.6 probability of being drawn (sum of them all) in any given draw. For five independent samplings to draw one of those four, you get a total probability of 0.6^5, which is approximately 0.0777. In other words, there is about a 7.77 percent chance that p_3 is missing — which means there is about a 92 percent probability of sampling it at least once. Particles with small importance weights will survive at a much lower rate than the ones with larger importance weights. This is exactly what you want to get from the resampling step.



New Particle Set

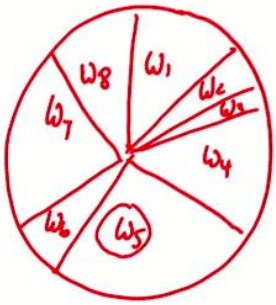
Question 14 (New Particle Set):

Modify the given algorithm to resample the list of particles and their respective importance weights. Take the list of particles and importance weights of *N* particles and sample it with replacement *N* times to get a new list of *N* particles; each particle should have a probability proportional to its importance weight for being picked *for each sample.*

Remember that you have already calculated the new particles and the corresponding importance weights; you now must construct a new particle set $p_3 = []$, so that the particles in p_3 are drawn according to their importance weights, w.

```
p2 = []
for i in range(N):
    p2.append(p[i].move(0.1, 5.0))
p = p2

w = []
for i in range(N):
    w.append(p[i].measurement_prob(Z))

p3 = []
p = p3
```

Question 15 (Resampling Wheel):

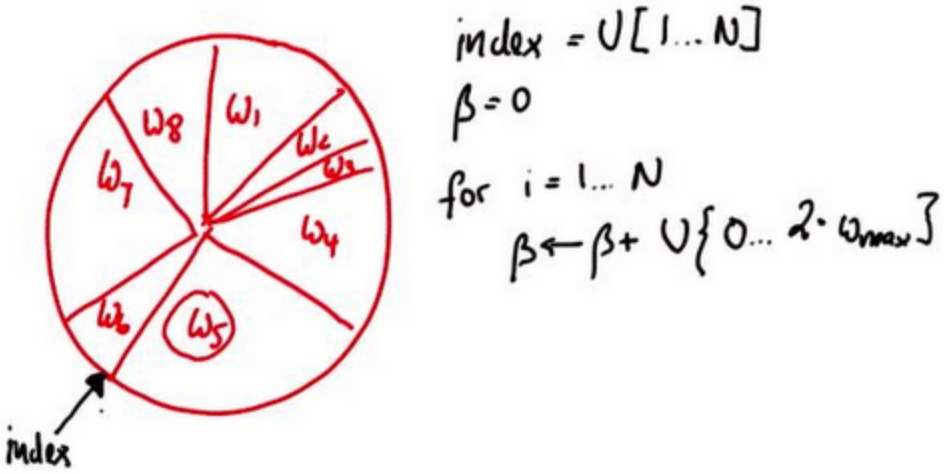


Represent all of the particles and importance weights in a big wheel. Each particle occupies a slice that corresponds to its importance weight. Particles with a bigger weight occupy more space, where particles with a smaller weight occupy less space.

Initially, guess a particle index, giving a uniform probability to each index from the set of all indices. This can be written as:

```
index = U[1...N]
```

We'll use a random selection of w_6 for this index for the visualization. The trick is now to construct a function, which you initialize to zero, and to which you add a uniformly drawn continuous value that sits between zero and $2*w_{max}$. (The variable w_{max} is the largest of the importance weights in the importance set.) We will call this function Beta (β).



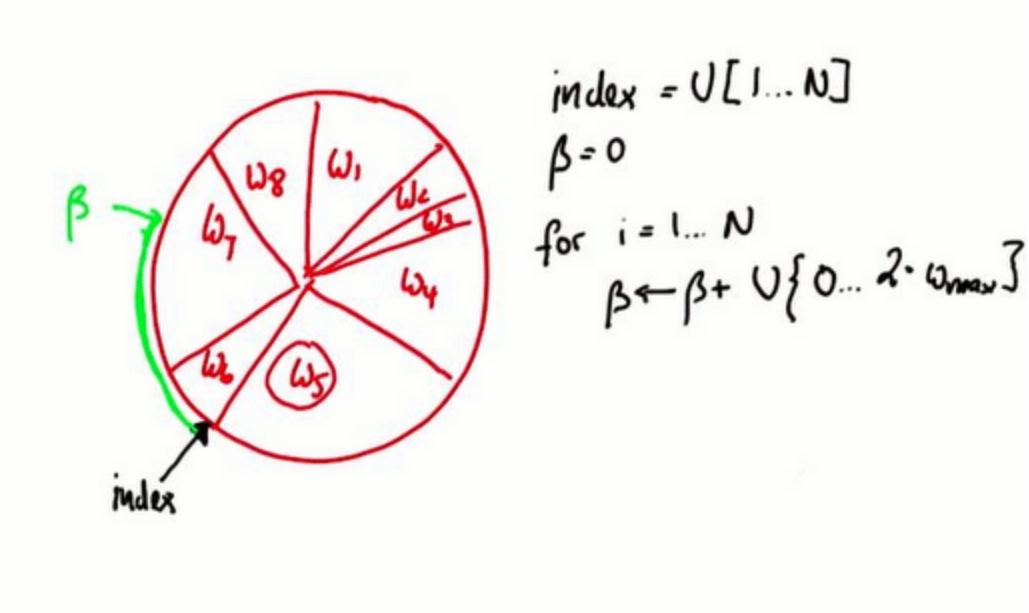
Since w_5 is the largest in this example, you are going to add a random value that might be as large as twice w_5. Suppose that you start at the position indicated in the picture above and add a β (randomly chosen between 0 and $2*w_{max}$) that brings you to w_7, as shown below. Now iterate the following loop; if the importance weights of the present particle doesn't suffice to reach all the way to the end of β, then subtract the value of the importance weight from β and add one to the index. This is written as:

```
while w[index] < Beta:
    Beta -= w[index]
    index += 1
```

What you have done is moved the index to the next *w* and removed a section of β.

By repeating this, you will eventually get to the point where beta is smaller than your w[index], then you pick the particle associated with that index. This particle will be added to your new list of particles!

When we do this *N* times, we get *N* particles, and we can see that particles will be chosen in proportion to their circumference on the circle.



Now your job is to implement this Resampling Wheel in Python!

Answer: Resampling Wheel (Sebastian's Code)

```
from math import *
import random

landmarks  = [[20.0, 20.0], [80.0, 80.0], [20.0, 80.0], [80.0, 20.0]]
world_size = 100.0

class robot:
    def __init__(self):
        self.x = random.random() * world_size
        self.y = random.random() * world_size
        self.orientation = random.random() * 2.0 * pi
        self.forward_noise = 0.0;
        self.turn_noise    = 0.0;
        self.sense_noise   = 0.0;

    def set(self, new_x, new_y, new_orientation):
        if new_x < 0 or new_x >= world_size:
            raise ValueError, 'X coordinate out of bound'
        if new_y < 0 or new_y >= world_size:
            raise ValueError, 'Y coordinate out of bound'
        if new_orientation < 0 or new_orientation >= 2 * pi:
            raise ValueError, 'Orientation must be in [0..2pi]'
        self.x = float(new_x)
        self.y = float(new_y)
        self.orientation = float(new_orientation)

    def set_noise(self, new_f_noise, new_t_noise, new_s_noise):
        # makes it possible to change the noise parameters
        # this is often useful in particle filters
        self.forward_noise = float(new_f_noise);
        self.turn_noise    = float(new_t_noise);
        self.sense_noise   = float(new_s_noise);

    def sense(self):
        Z = []
        for i in range(len(landmarks)):
            dist = sqrt((self.x - landmarks[i][0]) ** 2 + (self.y - landmarks[i][1]) ** 2)
            dist += random.gauss(0.0, self.sense_noise)
            Z.append(dist)
        return Z

    def move(self, turn, forward):
        if forward < 0:
            raise ValueError, 'Robot cant move backwards'
        # turn, and add randomness to the turning command
        orientation = self.orientation + float(turn) + random.gauss(0.0, self.turn_noise)
        orientation %= 2 * pi
```

```
        # move, and add randomness to the motion command
        dist = float(forward) + random.gauss(0.0, self.forward_noise)
        x = self.x + (cos(orientation) * dist)
        y = self.y + (sin(orientation) * dist)
        x %= world_size    # cyclic truncate
        y %= world_size
        # set particle
        res = robot()
        res.set(x, y, orientation)
        res.set_noise(self.forward_noise, self.turn_noise, self.sense_noise)
        return res

    def Gaussian(self, mu, sigma, x):
        # calculates the probability of x for 1-dim Gaussian with mean mu and var. sigma
        return exp(- ((mu - x) ** 2) / (sigma ** 2) / 2.0) / sqrt(2.0 * pi * (sigma ** 2))

    def measurement_prob(self, measurement):
        # calculates how likely a measurement should be
        prob = 1.0;

        for i in range(len(landmarks)):
            dist = sqrt((self.x - landmarks[i][0]) ** 2 + (self.y - landmarks[i][1]) ** 2)
            prob *= self.Gaussian(dist, self.sense_noise, measurement[i])
        return prob

    def __repr__(self):
        return '[x=%.6s y=%.6s orient=%.6s]' % (str(self.x), str(self.y), str(self.orientation))


def eval(r, p):
    sum = 0.0;
    for i in range(len(p)): # calculate mean error
        dx = (p[i].x - r.x + (world_size/2.0)) % world_size - (world_size/2.0)
        dy = (p[i].y - r.y + (world_size/2.0)) % world_size - (world_size/2.0)
        err = sqrt(dx * dx + dy * dy)
        sum += err
    return sum / float(len(p))

# --------

N = 1000
T = 10

myrobot = robot()

p = []
for i in range(N):
    r = robot()
    r.set_noise(0.05, 0.05, 5.0)#Sebastian's provided noise.
    p.append(r)

for t in range(T):
    myrobot= myrobot.move(0.1, 5.0)
    Z = myrobot.sense()

    p2 = []
```

```
    for i in range(N):
        p2.append(p[i].move(0.1, 5.0))
    p = p2

    w = []
    for i in range(N):
        w.append(p[i].measurement_prob(Z))

    p3 = []
    index = int(random.random() * N)
    beta = 0.0
    mw = max(w)
    for i in range(N):
       beta += random.random() * 2.0 * mw
       while beta > w[index]:
           beta -= w[index]
           index = (index + 1) % N
       p3.append(p[index])
    p = p3

    print eval(myrobot, p)

if eval(myrobot, p) > 15.0:
    for i in range(N):
        print '#', i, p[i]
    print 'R', myrobot
```

Question 16 (Orientation):

Will orientation never play a role?

1. Yes
2. No, eventually they matter

Answer: Orientation

- b. No, eventually they matter

Orientation does matter in the second step of particle filtering because the prediction is so different for different orientations.

Programing the Orientation

Program the particle filter to run twice:

Solution:

- N = 1000 T = 2

```
    myrobot = robot()

    p = []
    for i in range(N):
        r = robot()
        r.set_noise(0.05, 0.05, 5.0)
        p.append(r)

    for t in range(T): # insert a for loop, indent everything below until print p
        myrobot= myrobot.move(0.1, 5.0)
        Z = myrobot.sense()

    ...

    print p # only print the final distribution
```

When you run this code the orientations are not that worked out.

What if you move ten steps forward:

```
    N = 1000
    T = 10 # robot moves 10 steps forward instead of 2

    myrobot = robot()

    p = []
    for i in range(N):
        r = robot()
        r.set_noise(0.05, 0.05, 5.0)
        p.append(r)

    for t in range(T):
        myrobot= myrobot.move(0.1, 5.0)
        Z = myrobot.sense()

    ...

    print p # only print the final distribution
```

When you run this code, you get orientations that all look alike, the orientations are all between 3.6 and 3.9, the y's are all between 53 and 55, and the x's are all around 39. This consistency is how you know the particle filter is working.

Programming

Rather than print out the particles themselves, you can print out the overall quality of the solution. To do this you already have an eval code that takes in the robot position, r, and a particle set, p, to compute the average error of each particle relative to the robot position in *x* and *y*, but which doesn't consider orientation. The way the function works is that it compares the *x* and *y* of the particle with the *x* and *y* of the robot and computes the Euclidean distance with the *x* and *y* distances and then averages all of those values.

```
    def eval(r, p):
        sum = 0.0;
        for i in range(len(p)): # calculate mean error
            # the last part of this next line is normalization for a cyclical worl
            dx = (p[i].x - r.x + (world_size/2.0)) % world_size - (world_size/2.0)
            dy = (p[i].y - r.y + (world_size/2.0)) % world_size - (world_size/2.0)
            err = sqrt(dx * dx + dy * dy)
            sum += err
        return sum / float(len(p))
```

Take the function eval and produce a sequence of performance evaluations so that when you hit the run button you return error numbers that look something like this:

```
4.90551267551
3.6796887545
2.98309363912
2.84573149608
3.1058375807
3.32277284382
3.27281592582
3.1155504812
2.77833986902
2.37397382004
```

Solution:

```
print eval(myrobot, p)
```

Looks simple! Remember that when you print this statement over and over you do not always get the same results.

What You Programmed (You and Sebastian)

You just programmed a full particle filter! Sebastian provided you with a very primitive robot simulator that uses landmarks as a way of taking measurements and uses three-dimensional robot coordinates (x, y, and orientation). You solved the estimation problem in just 30 lines of code! This is a small amount of code for something that is amazingly powerful. You can reuse these lines of code in pretty much all problems you might study that require particle filters.

Recap: The Math Behind It All

For your particle filters you had two kinds of updates:

- 1. Measurement updates

For the measurement update you computed posterior over state, given a measurement update, that was proportional to the normalization of the probability of the measurement, given the state, multiplied by $P(x)$. This is written as:

- $P(X|2) \alpha P(2|X) P(X)$

Let's flesh out this formula to show you how you used it. The distribution, $P(X)$, was your set of particles; 1000 particles together represented your $P(X)$-- prior X. The importance weight is represented by $P(2|X)$. Technically speaking, the particles with the importance weights are a representation of the distribution. But we want to get rid of the importance weights so by resampling you work the importance weight back into the set of particles so that the resulting particles $P(X|2)$ will represent the correct posterior.

- 2. Motion updates

In the motion update you computed a posterior over distribution one time step later, and that is the convolution of the transition probability, multiplied by the prior. This is written as:

- $P(X') = \Sigma P(X'|X)P(X)$

Similarly, let's have a look at how this formula is operating within your particle filter. Your set of particles is written as $P(X)$, and your sample from the sum, Σ. By taking a random particle from $P(X)$ and applying the motion model with the noise model to generate a random particle (X^1). As a result you get a new particle set, $P(X^1)$ that is the correct distribution after the robot motion.

This math is actually the same for all of the filters we have talked about so far!

Question 15 (Filters):

Which filters did Sebastian use in his job talk at Stanford?

- Histogram filters
- Kalman filters
- Particle filters
- None

Answer: Filters

1. Histogram filters (1998) c. Particle filters (2003)

2012

The difference between the Google car and what you have learned so far, is that the Google car follows a bicycle model and the sensor data. Instead of using landmarks like the robot, the Google car uses a really elaborate road map. It takes a single snapshot, matches it to the map and the better the match, the higher the score. Additional sensors, like GPS and inertial sensors, also differentiate the Google car from your robot model.

Despite these differences, you do have a solid understanding of the gist of how the Google car is able to understand where it is and where other cars are. When you build a system, you have to dive into more elaborate systems, which is doable.