The Kalman filter solved a significant problem for us: it allowed us to deal with continuous spaces by defining our "belief" as a Gaussian distribution. However, what if that's a bad assumption? What if a Gaussian is a bad idea? For example, consider Figure 1.

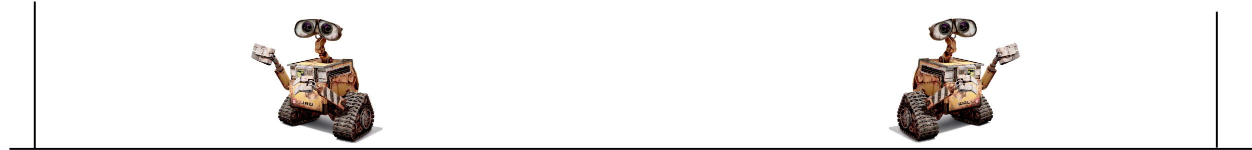


Figure 1: Two possible locations for the robot

In this, our robot is moving back and forth between the two walls. Assume it has a sensor with low error. After taking a measurement of distance d from a wall, this means that the robot could be in two different locations, each of which is d meters from one of the walls, in the two locations illustrated.

Is there a single Gaussian that can capture this belief? Ideally, we'd like to use TWO Gaussians, but any single Gaussian would have to be centered in the middle and very wide to capture that these two locations are equally likely; this is obviously incorrect. So, we're dissatisfied.

What we'd like to do, is represent ANY distribution, not just Gaussians. However, most distributions we could draw don't have an easy mathematical representation.

The key trick, then, is this: we'll represent our belief at any given time with a collection of samples drawn from that distribution.

Sampling From a Distribution

When we say we sample from a distribution, we mean that we choose some discrete points, with likelihood defined by the distribution's probability density function. For example, in Figure 2, we can see samples drawn from the two illustrated distributions. The density of these points will approximate the probability density function of the distribution; the larger the number of points, the better the approximation.

A single sample p drawn from a distribution $p(x)$ is denoted $p \sim p(x)$.

Therefore, with enough samples, we don't NEED to mathematically define the distribution (the blue line). Instead, we can just use the density of the red points.

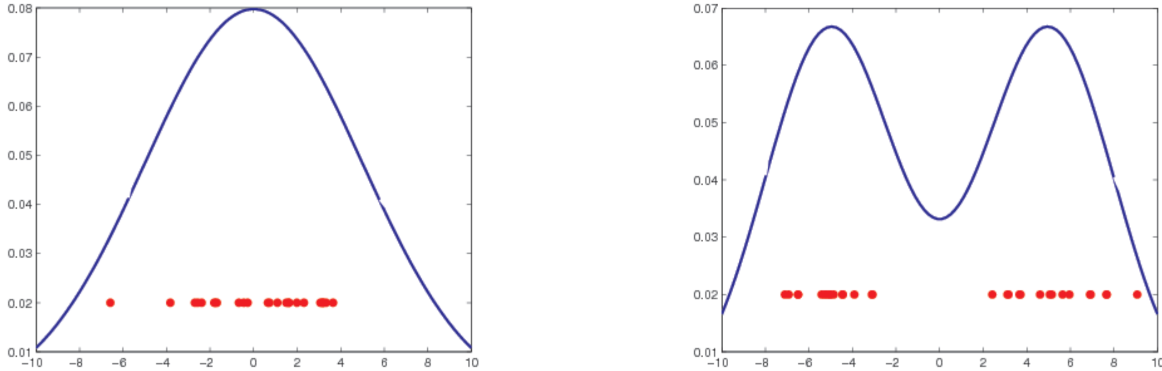


Figure 2: Samples (red dots) drawn from two different distributions.

Sidebar on sampling from distributions: Usually when we generate random numbers, we do it by sampling from a uniform distribution (all numbers are equally likely). As we get deeper into applications of probability, we often find we want to generate numbers from a non-uniform distribution, such as a gaussian, where numbers closer to the mean are more likely. There are two ways we'll talk about doing this - inverse transform sampling, and rejection sampling.

Before we talk about inverse transform, we need to talk about cumulative distribution functions (cdf). They are like regular distributions, but represent the probability of getting that value or a value less than it, $F(x) = \int_{-\infty}^{x} f(t)dt$. Note that when the density function is high, the slope of the cumulative distribution is high, and when the density function is low, the slope of the cumulative distribution is low. This function is always increasing, which is necessary if we're going to invert it. Below is the pdf and cdf for three gaussians.

Once we have this, all we do is take the inverse, generate a random number from a uniform distribution, feed it through the inverse cdf, and we have our number. In regions where the slope of the cdf is high, those numbers are likely to be generated. The disadvantage of this is that it can be difficult to compute the cdf, and it can be difficult to invert the cdf as well.

For rejection sampling we take the pdf $f(x)$ and generate two numbers x, y from a uniform distribution. If $f(x) \leq y$ then a keep x as a sample, otherwise we reject it. In regions where the pdf is high, we are less likely to reject an x, and so we will get more values in that region. The disadvantage is that if the distribution has narrow spikes, we can spend a lot of time rejecting samples.

The Particle Filter

The idea behind the particle filter is similar to the others, there will be a prediction step, and then an update step. The difference is that instead of keeping track of a complete

distribution or pdf, we will *approximate* the pdf with a collection of points, called particles.

Let's use our example again of a robot in a 1D world, with a single wall at x-value 0 on the left, and a rangefinder.

We'll start with some large number of points, randomly distributed in a way that reflects our prior. For Kalman Filters, this had to be a Gaussian, which is difficult to manage when you don't know ANYTHING. For Particle Filters, they can be evenly and randomly distributed across the entire space. Each of the particles can be viewed as a candidate position for our robot. This can be seen in Figure 3a.

Algorithm 1 Initializing particles

for all p in particles **do**
  p.location $\leftarrow$ random location
  p.weight $\leftarrow$ 1 ▷ Weight, for now, is irrelevant
end for

We then move each particle as if it was the robot. Now, suppose we told our robot to move 1 meter to the right. Now, remember if the observed distance moved is u, the amount moved x is $x = b \cdot u + \mathcal{N}(0, \sigma)$ for some b and σ. Suppose in this case, b is 1, and $\sigma = 0.5$.

So, for each particle, we draw a separate "real amount moved" from the distribution $\mathcal{N}(1, 0.5)$, and move it that far. That gives us Figure 3b. It may be difficult to see, but each has shifted over a different amount. This is how we handle taking actions.

Algorithm 2 Moving particles u meters

for all p in particles **do**
  dx $\leftarrow b \cdot u + \mathcal{N}(0, \sigma)$ ▷ Different dx for each particle
  p.location $\leftarrow$ p.location + dx
end for

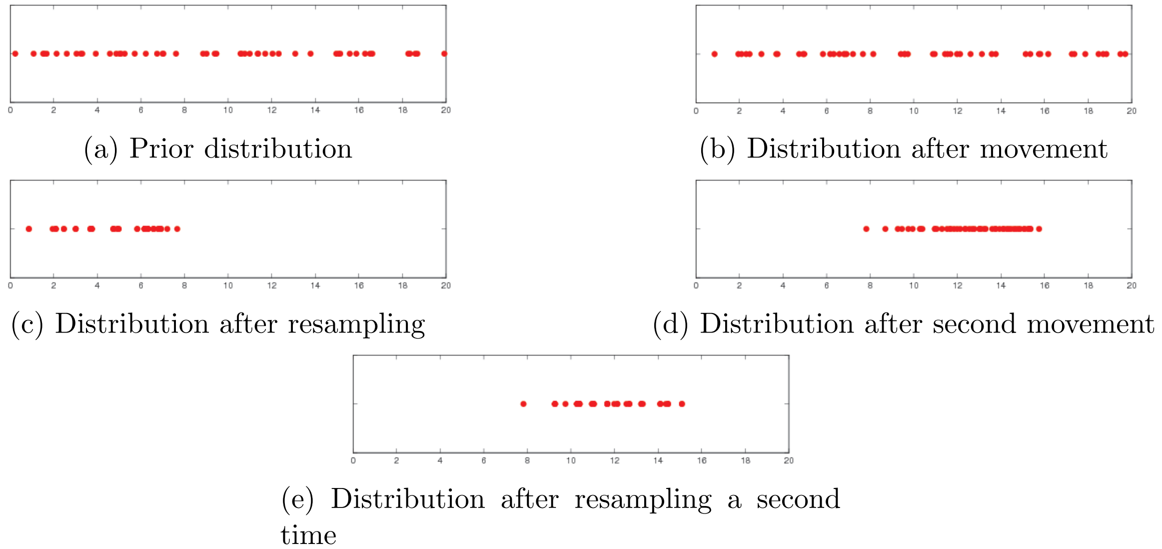
We then take a reading. Now, our rangefinder again gives us an observation $o = c \cdot x + \mathcal{N}(0, \sigma)$. We'll give it a c of 1, and a σ of 3, and say that our observation is that the distance to the wall is 4.

For each particle, we now ask, if the robot were exactly where the particle is, and had a perfect sensor, what measurements would it have made? Once we have that, for each particle we calculate the probability of getting observation o given the theoretical actual measurement of o_p. This becomes the "weight" of the particle.

So in our scenario, o is 4, σ is 3, and o_p happens to be the same as the robot's location (since the wall is at location 0). So, particles close to 4 will have a larger probability than particles far away from 4.

We then *resample* from our particles, where the likelihood of each particle being chosen is equal to the calculated probability of that particle over the sum of all weights. Likely particles will be sampled multiple times, making some duplicates in the resampled set, and unlikely particles may not be resampled at all. This set can be seen in Figure 3c.

This finishes one loop of the Particle Filter. Let's do a second loop. First we'll move our robot 8 meters to the right. Remember, each particle moves a slightly different amount, so



(a) Prior distribution

(b) Distribution after movement

(c) Distribution after resampling

(d) Distribution after second movement

(e) Distribution after resampling a second time

Figure 3: Samples before movement, after movement, and after sensing.

all of our repeated samplings from the previous step will be spread out. This can be seen in Figure 3d.

We then get an observation of 14 meters. Again, particles are resampled, with probability $\mathcal{N}(14, 3)$, giving us Figure 3e.

The Algorithm

For our one-dimensional problem, with a single landmark that can be sensed, we therefore have the following Particle Filter algorithm. Let `particles` be a list of N particles, each of which have a randomly-assigned location.

Then, take an action u, and move each particle by a different amount close to u, sampled from the motion model. Then, make an observation o, and weight each particle by how likely it is to have made that same measurement. Finally, resample from your particles, such that heavily weighted particles are more likely to be resampled. Repeat as long as wanted.

The cluster of particles will represent the possible locations of the robot. After each step, you can query for the most heavily-weighted particle to use as a most-likely location of the robot.

The Awesome Mathematical Property

A particle filter is a pretty intuitive approach. But, it turns out that it's way better than just intuitive. Let's consider the alternative approach.

We *could* work very, very hard to keep track of an arbitrarily-shaped distribution to model $p(x_t|o_{1:t}, u_{1:t})$ (the probability of the robot being in a given location given the sequence of

Algorithm 3 Weighting and resampling

```
weightsum=0
for all p in particles do
    o_p ← theoretical perfect measurement from p.location
    p.weight ← pdf(o − c · o_p, 0, σ)        ▷ The probability of our sensor being wrong by o − c · o_p.
    weightsum ← weightsum + p.weight
end for
for all 1≤i≤N do
    r ← weightsum * rand()        ▷ Random number between 0 and sum
    samplesum← 0
    for all p in particles do
        samplesum← samplesum+p.weight
        if r ≤ samplesum then        ▷ Heavily weighted particles are more likely to cause this
            newParticles[i].location←p.location
            break
        end if
    end for
end for
particles ← newParticles        ▷ New, resampled list of particles
```

all previous actions and observations). That would be a complicated probability density function, but if we could do it, it would be spectacularly informative.

BUT, it turns out that if we use the above particle filter algorithm, the likelihood that a given state hypothesis x_t exists in our particle set is *exactly* the same as this probability! It not only makes intuitive sense, but it also mathematically unassailable.