Lesson 1: Localization

Intro

In this course you will learn how to program self-driving cars! Specifically, in this unit, you will learn how to program a localizer, which is a directional aid used to assist machines in making informed navigation decisions.

Over the past decade computer scientists have developed methods using sensors, radars and software to program vehicles with the ability to sense their own location, the locations of other vehicles and navigate a charted course.

Stanford University Professor and director of the Stanford Artificial Intelligence Lab, Sebastian Thrun's autonomous cars, Stanley and Junior illustrate the progress that has been made in this field over the last decade. Additionally, the Google Driverless Car Project seeks to further research and development to make driverless cars a viable option for people everywhere. Driverless cars sound neat, huh? Want to program your own?

Let's get started!

The Problem of Localization

Localization is the ability for a machine to locate itself in space.

Consider a robot lost in space. Within its environment, how can the robot locate where it is?

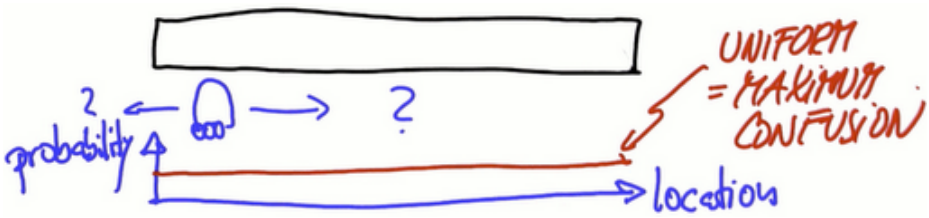
Rather than install a GPS device in our robot, we are going to write a program to implement localization.

Our localization program will reduce the margin of error considerably compared to a GPS device, whose margin of error can be as high as ten meters. For our robot to successfully and accurately navigate itself through space, we are looking for a margin of error between two and ten centimeters.

Imagine a robot resides in a one-dimensional world, so somewhere along a straight line, with no idea where it is in this world. For an example of such a world, we can imagine a long, narrow hallway where it is only possible to move forward or backwards; sideways motion is impossible.

Since our robot is completely clueless about its location, it believes that every point in this one dimensional world is equally likely to be its current position. We can describe this mathematically by saying that the robot's probability function is uniform (the same) over the sample space (in this case, the robot's one-dimensional world).

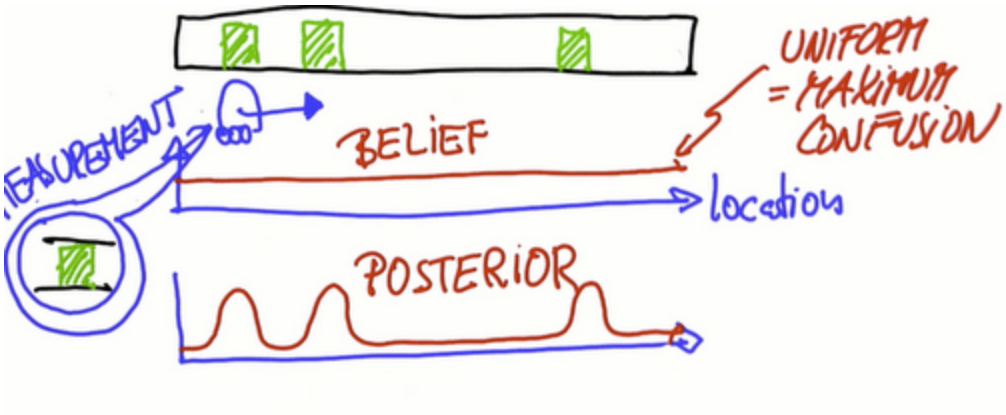
If we were to draw a graph of this probability function with probability on the vertical axis and location on the horizontal axis, we would draw a straight, level line. This line describes a uniform probability function, and it represents the state of maximum confusion.



Assume there are three landmarks, which are three doors that all look alike and we can distinguish a door from a non-door area.

If the robot senses it is next to a door, how does this affect our belief — or the probability that the robot is near a door?

In the new function, there are three bumps aligned with the location of the doors. Since the robot has just sensed that it is near a door, it assigns these locations greater probability (indicated by the bumps in the graph) whereas, all of the other locations have decreased belief.



This function represents another *probability distribution*, called the *posterior* belief where the function is defined after the robot's sense measurement has been taken.

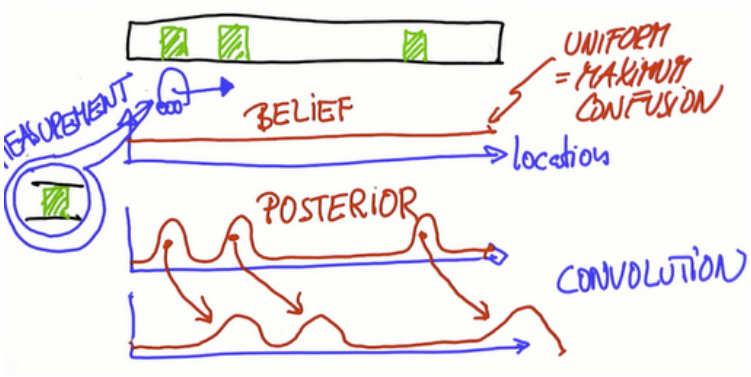
The *posterior* function is the best representation of the robot's current belief, where each bump represents the robot's evaluation of its position relative to a door.

However, the possibility of making a bad measurement constantly looms over robotics, and over the course of this class we will see various ways to handle this problem.

Robot Movement

If the robot moves to the right a certain distance, we can shift the belief according to the motion.

Notice that all of the bumps also shift to the right, as we would expect. What may come as a surprise, however, is that these bumps have not shifted perfectly. They have also flattened. This flattening is due to the uncertainty in robot motion: since the robot doesn't know exactly how far it has moved, its knowledge has become less precise and so the bumps have become less sharp.

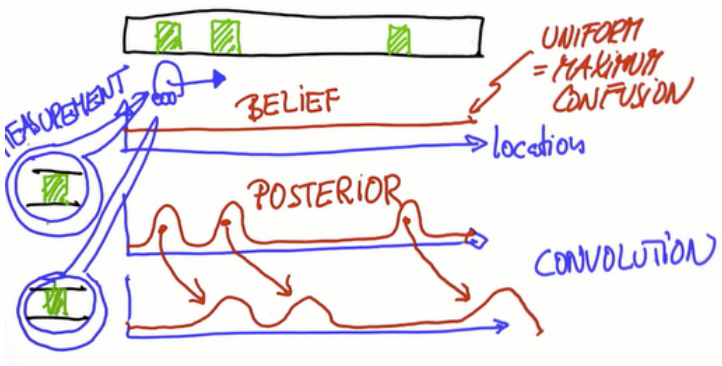


When we shift and flatten these bumps, we are performing a convolution. Convolution is a mathematical operation that takes two functions and measures their overlap. To be more specific, it measures the amount of overlap as you slide one function over another. For example, if two functions have zero overlap, the value of their convolution will be equal to zero.

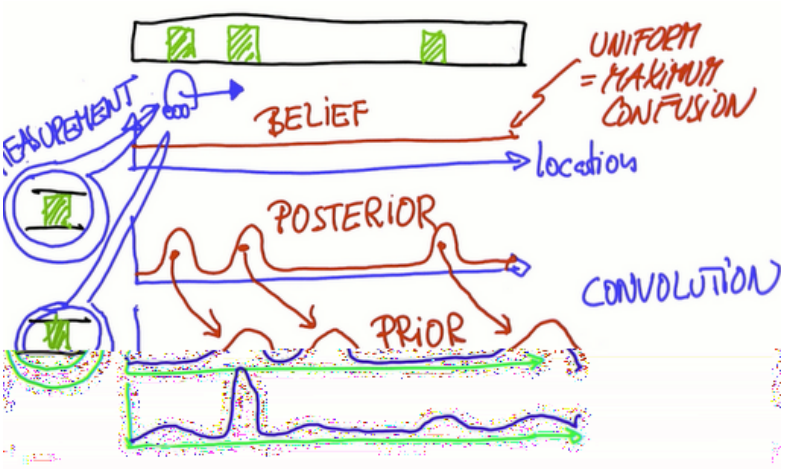
If they overlap completely, their convolution will be equal to one. As we slide between these two extreme cases, the convolution will take on values between zero and one. The animations here and here should help clarify this concept.

In our convolution, the first function is the belief function (labeled "posterior" above), and the second is the function which describes the distance moved, which we will address in more depth later. The result of this convolution is the shifted and flattened belief function shown below.

Now, assume that after the robot moves it senses itself right next to a door again so that the measurement is the same as before. Just like after our first measurement, the sensing of a door will increase our probability

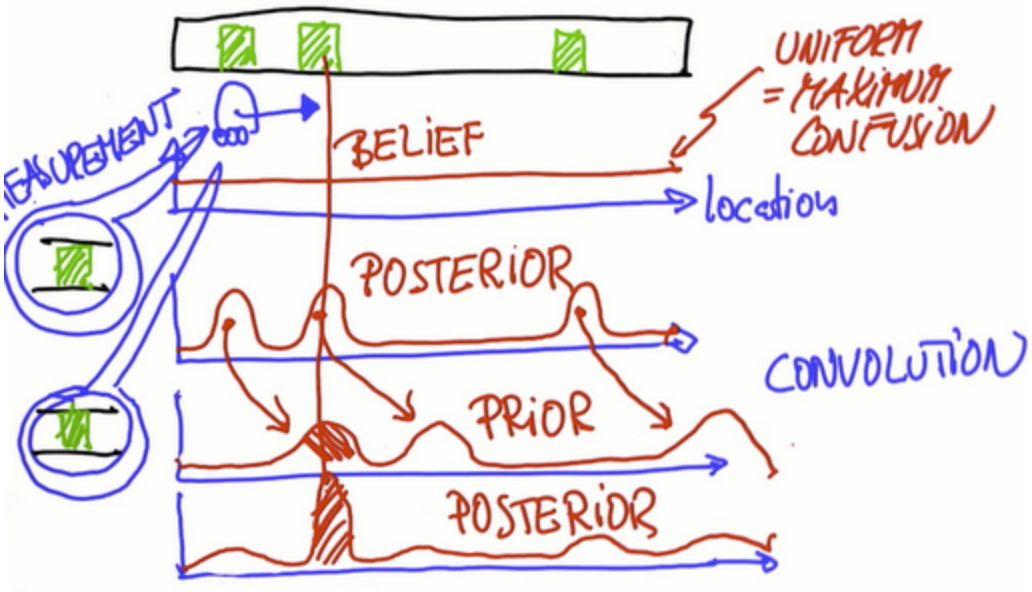


function by a certain factor everywhere where there is a door. So, should we get the same posterior belief that we had after our first measurement? No! Because unlike with our first measurement, when we were in a state of maximum uncertainty, this time we have some idea of our location prior to sensing. This prior information, together with the second sensing of a door, combine to give us a new probability distribution, as shown in the bottom graph below.



In this graph we see a few minor bumps, but only one sharp peak. This peak corresponds to the second door. We have already explained this mathematically, but let's think intuitively about what happened. First, we saw the first door. This led us to believe that we were near *some* door, but we didn't know *which*. Then we moved and saw *another* door. We saw two doors in a row! So of course our probability function should have a major peak near the only location where we would expect to see two doors in a row.

Once again, it's important to note that we still aren't certain of our location, but after two measurements we are more sure than we were after one or zero measurements. What do you think would happen as we made more and more measurements?



Congratulations! You now understand both *probability* and *localization*! The type of localization that you have just learned is known as Monte Carlo localization, also called *histogram filters*.

Exercises

Question 1 (Uniform Probability Quiz)

If a robot can be in one of five grid cells, labeled X_i ,for i=1...5, what is the probability for each X_i ?

Answer 1 (Uniform Probability Quiz)

$P(X_i)=0.2$

Question 2 (Uniform Distribution Quiz)

Modify the empty list:

```
p = []
```

such that p becomes a uniform distribution over five grid cells, expressed as a vector of five probabilities. For example, the probability that the robot is in cell two, can be written as:

```
p[2] = 0.2
```

and given that each cell has the same probability that the robot will be in there, you can also write the probability that the robot is in cell five by writing:

```
p[5] = 0.2
```

For a refresher on python, the links here will be helpful.

Answer 2 (Uniform Distribution Quiz)

A simple solution is to specify each element in the list p.

```
p =  [0.2, 0.2, 0.2, 0.2, 0.2]

print p
```

```
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Question 3 (Generalized Uniform Distribution)

Modify your code to create probability vectors, p of arbitrary size, n. For example use:

```
n=5
```

to help us to verify that our solution matches the previous solutions.

Answer 3 (Generalized Uniform Distribution)

Use a for loop:

```
#define global variables p and n.
p = [ ]
n = 5
#Use a for loop to search for a cell, i, in a world with a given
n

for i in range (n)
#Append to the list ''n'' elements each of size one over ''n''. R
to use floating point numbers

    p.append(1./n)

print p
```

If we forget to use floating point operations, python will interpret our division as integer division. Since 1 divided by 5 is 0 with remainder 1, our incorrect result would be [0,0,0,0,0]

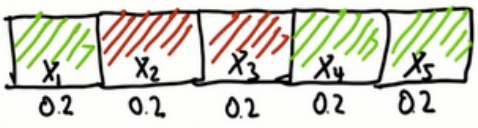
Now we are able to make p a uniform probability distribution regardless of the number of available cells, specified by n.

Question 4 (Probability After Sense)

Now that we can establish the robot's initial belief about its location in the world, we want to update these beliefs given a measurement from the robot's sensors.

Examine the measurement of the robot in a world with five cells, x1 through x5.

Say we know that externally two of the cells (x2 and x3) are colored red, and the other three (x1, x4 and x5) are green. We also know that our robot senses that it resides in a red cell.



How will this affect the robot's belief in its location in the world?

Knowing that the robot senses itself in a red cell, let's update the belief vector such that we are more likely to reside in a red cell and less likely to reside in a green cell.

To do so, come up with a simple rule to represent the probability that the robot is in a red or a green cell, based on the robot's measurement of 'red':

```
red cells * 0.6
green cells * 0.2
```

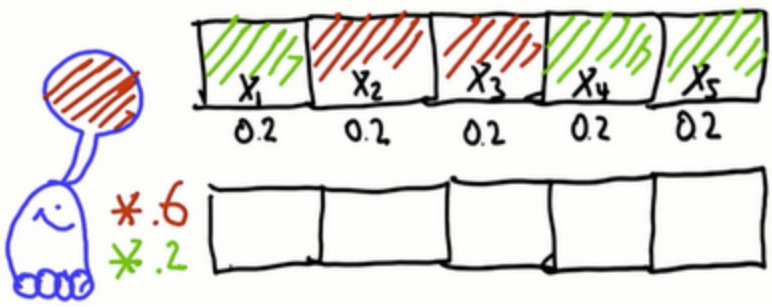
For now, we have chosen these somewhat arbitrarily, but since 0.6 is three times larger than 0.2, this should increase the probability of being in a red cell by something like a factor of three.

Keep in mind that it is possible that our sensors are incorrect, so we do not want to multiply the green cell probability by zero.

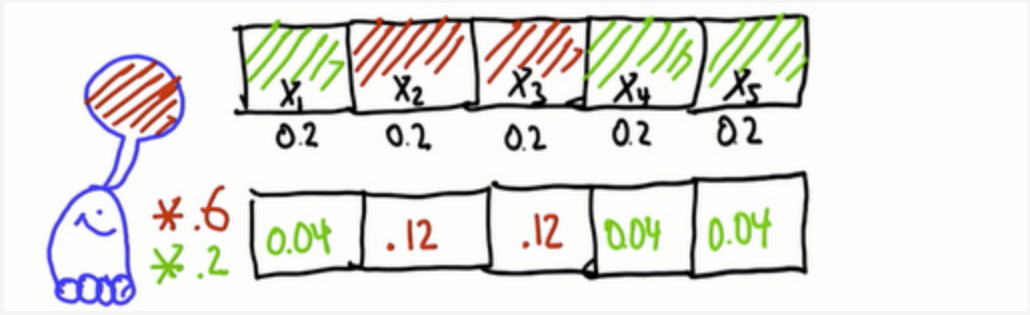
What is the probability that the robot is in a red cell and the probability that it's in a green cell?

Answer4 (Probability After Sense)

Find the probability for each cell by multiplying the previous belief (0.2 for all cells) by the new factor, whose value depends on the color of the cells.




```
For red cells, 0.2*0.6 = .12
For green cells 0.2*0.2 = 0.04
```



So are these our probabilities?

Question 5 (Compute Sum)

No! These are not our probabilities, because they do not add up to one. Since we know we are in *some* cell, the individual probabilities must add up to one. What we have now is called an *unnormalized probability distribution*

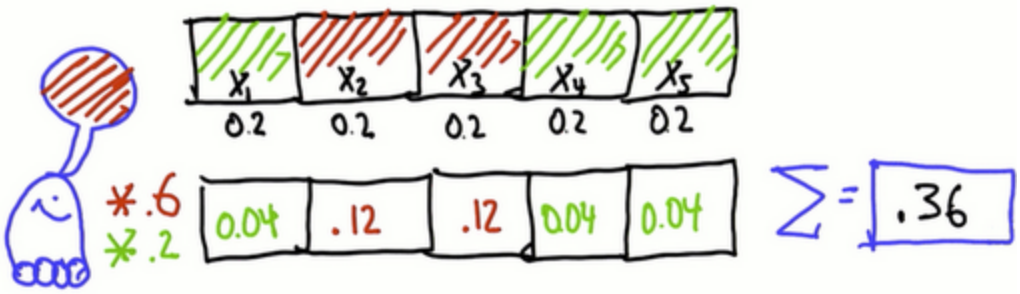
So, how do we *normalize* our distribution?

First, find the current sum of the individual probabilities. Then, find a proper *probability distribution*. A probability distribution is a function that represents the probability of any variable, *i*, taking on a specific value.

Compute the sum of the values to see if you have a proper probability distribution — the sum should be one.

Answer 5 (Compute Sum)

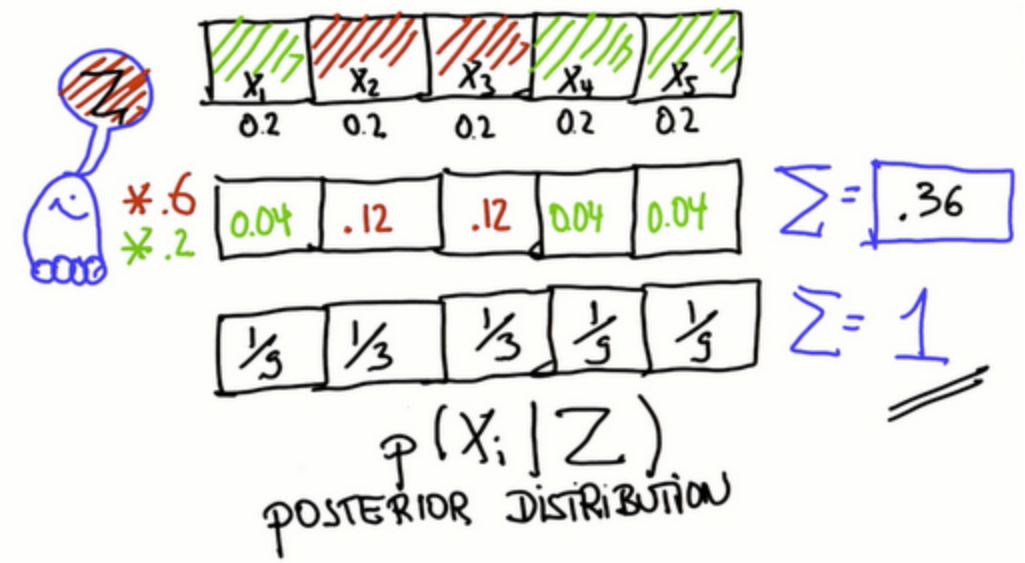
Since the sum of the cells is not one, our updated distribution is not a proper probability distribution.



Question 6 (Normalize Distribution)

To obtain a probability distribution, divide each number in each box by the sum of the values, 0.36.

Answer 6 (Normalize Distribution)



- Green cells = 1/9
- Red cells= 1/3

This is a probability distribution written as: $P(X_i|Z)$

and read as: The *posterior distribution* of place X_i given measurement Z, Z being the measurement from the robot's sensors. This is an example of conditional probability. If you need more help on this subject, this video may help. |

Question 7 (pHit and pMiss)

Starting with our state of maximum confusion

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
```

We want to write a program that will multiply each entry by the appropriate factor: either pHit or pMiss.

We can start off by not worrying about whether this distribution will sum to one (whether it is *normalized*). As we showed in the last few questions, we can easily normalize later.

Write a piece of code that outputs p after multiplying pHit and pMiss at the corresponding places.

- pHit = 0.6 → factor for matching measurement
- pMiss = 0.2 → factor for non-matching measurement

Answer 7 (pHit and pMiss)

One way to do this is to go through all five cases and manually multiply the pMiss or pHit case:

```
p = [0.2, 0.2, 0.2, 0.2, 0.2)
pHit = 0.6
pMiss = 0.2
p[0] = p[0]*pMiss
p[1] = p[1]*pHit
p[2] = p[2]*pHit
p[3] = p[3]*pMiss
p[4] = p[4]*pMiss
print p

0.04000000000000004,        0.12,        0.12,        0.040000000000
0.040000000000000008
```

Question 8 (Sum of Probabilities)

We know that this is not a valid distribution because it does not sum to one. Before we normalize this distribution we need to calculate the sum of the individual probabilities.

Modify the program so that you get the sum of all the p's.

Answer 8 (Sum of Probabilities)

Use the Python sum function:

```
p = [0.2, 0.2, 0.2, 0.2, 0.2)
pHit = 0.6
pMiss = 0.2
p[0] = p[0]*pMiss
p[1] = p[1]*pHit
p[2] = p[2]*pHit
p[3] = p[3]*pMiss
p[4] = p[4]*pMiss
print sum(p)

0.36
```

Question 9 (Sense Function)

Let's make our code more elegant by introducing a variable, world, which specifies the color of each of the cells — red or green.

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
#introduce variable '''world'''
world = ['green', 'red', 'red', 'green', 'green']
pHit = 0.6
pMiss = 0.2
```

Furthermore, let's say that the robot senses that it is in a red cell, so we define the measurement Z to be red:

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
world = ['green', 'red', 'red', 'green', 'green']

#define the measurement ''Z'' to be red
Z = 'red'
pHit = 0.6
pMiss = 0.2
```

Define a function to be the measurement update called sense, which takes as input the initial distribution p and the measurement Z.

Enable the function to output the non-normalized distribution of q, in which q reflects the non-normalized product of input probability (0.2) with pHit or pMiss in accordance with whether the color in the corresponding world cell is red (hit) or green (miss).

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]'''
world = ['green', 'red', 'red', 'green', 'green']
Z = 'red'
pHit = 0.6
pMiss = 0.2
def sense(p, Z):
    #Define a function '''sense'''

    return q
print sense(p, Z)
```

When you return q, you should expect to get the same vector answer as in question 6, but now we will compute it using a function. This function should work for any possible Z (red or green) and any valid p[]

Answer 9 (Sense Function)

Step 1: One way to do this is as follows:

```
p = [0.2, 0.2, 0.2, 0.2, 0.2)
world = ['green', 'red', 'red', 'green', 'green']
Z = 'red'
pHit = 0.6
pMiss = 0.2

def sense(p, Z):
    q = [ ]
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))
    return q

print sense(p, Z)
```

This code sets hit equal to one when the measurement is the same as the current entry in world and 0 otherwise. The next line appends to the list q[ ] an entry that is equal to p[i]*pHit when hit = 1 and p[i]*pMiss when hit = 0.

This gives us our non-normalized distribution. A "binary flag" refers to the following piece of code: (hit * pHit + (1-hit) * pMiss). The name comes from the fact that when the term (hit) is 1, the term (1-hit) is zero, and vice-versa.

Question 10 (Normalized Sense Function)

Modify this code so that it normalizes the output for the function sense and adds up to one.

Answer 10 (Normalized Sense Function)

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
world = ['green', 'red', 'red', 'green', 'green']
Z = 'red'
pHit = 0.6
pMiss = 0.2
def sense(p, Z):
    q = [ ]
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))

    #First, compute the sum of vector q, using the sum function
    s = sum(q)
    for i in range (len(p)):
    # normalize by going through all the elements in q and divide
        q[i]=q[i]/s
```

```
    return q

print sense(p, Z)

[0.1, 0.3, 0.3, 0.1, 0.1]
```

Question 11 (Test Sense Function)

Try typing *green* into your Z, sensory measurement variable, and re-run your code to see if you get the correct result.

Answer 11 (Test Sense Function)

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
world = ['green', 'red', 'red', 'green', 'green']
#Make 'green' your Z, measurement variable, and re-run your code
if you get the correct result
Z = 'green'
pHit = 0.6
pMiss = 0.2
def sense(p, Z):
    q = [ ]
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))

    s = sum(q)
    for i in range (len(p)):
        q[i] = q[i]/s
    return q
print sense(p, Z)

[0.27,  0.09, 0.09, 0.27, 0.27]
```

This output looks good. Now the green cells all have higher probabilities than the red cells. The "division by 44" referred to in the video comes from the normalization step where we divide by the sum. The sum in this case was 0.44.

Question 12 (Multiple Measurements)

Modify the code in such a way that there are multiple measurements by replacing Z with a measurements vector.

Assume the robot is going to sense red, then green.

Can you modify the code so that it updates the probability twice and gives you the posterior distribution after both measurements are incorporated so that any sequence of measurement, regardless of length, can be processed?

Answer 12 (Multiple Measurements)

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
world = ['green', 'red', 'red', 'green', 'green']
#Replace Z with measurements vector and assume the robot is go
sense red, then green
measurements = ['red', 'green']
pHit = 0.6
pMiss = 0.2
def sense(p, Z):
    q = [ ]
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))
    s = sum(q)
    for i in range (len(p)):
        q[i] = q[i]/s
    #As often as there are ''measurements'' use the following for
    return q
for k in range(len(measurements)):
    #Grab the k element, apply to the current belief, p, and upda
belief into itself
    p = sense(p, measurements [k])
    print p
    #run this twice and get back the uniform distribution:
```

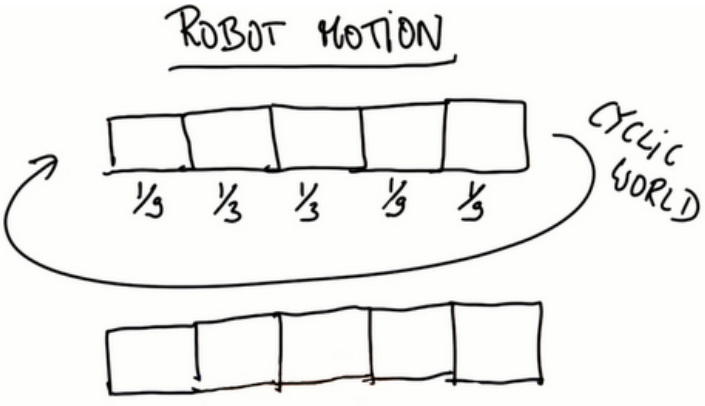
This code defines the function sense and then calls that function once for each measurement and updates the distribution.

```
[0.2, 0.2, 0.2, 0.2, 0.2]
```

Exact Motion

Suppose there is a distribution over the cells such as:

$$\frac{1}{9}\ \frac{1}{3}\ \frac{1}{3}\ \frac{1}{9}\ \frac{1}{9}$$

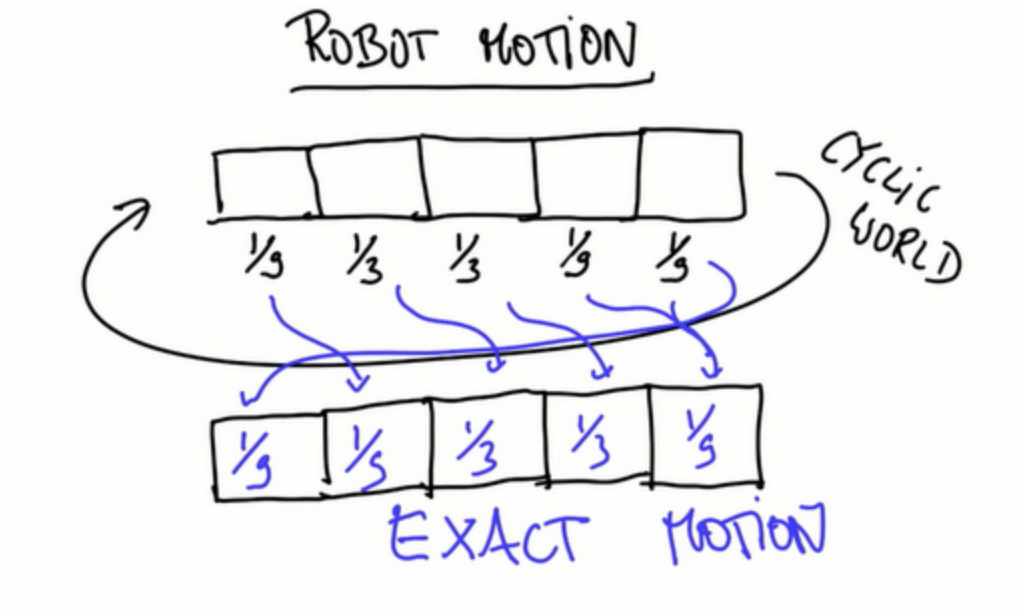


We know the robot moves to the right, and we will assume the world is cyclic. So, when the robot reaches the right-most cell, it will circle around back to the first, leftmost cell.

Question 1 (Exact Motion)

Assuming the robot moves exactly one position to the right, what is the posterior probability distribution after the motion?

Answer 1 (Exact Motion)



Everything shifts to the right one cell.

Question 2 (Move Function)

Define a function move with an input distribution p and a motion number U, where U is the number of grid cells moving to the right or to the left.

Program a function that returns a new distribution q, where if U == 0, q is the same as p.

- If U == 1, all the values cyclically shift to the right by 1
- If U == 3, all the values cyclically shift to the right by 3
- If U == -1, all the values cyclically shift to the left by 1

Changing the probability for cell 2 from zero to one will allow us to see the effect of the motion.

```
p = [0, 1, 0, 0, 0)

world = ['green', 'red', 'red', 'green', 'green']
measurements = ['red', 'green']
pHit = 0.6
```

```
pMiss = 0.2
def sense(p, Z):
    q = []
    for i in range(len(p)):
        hit = (Z == world[i])
        q.append(p[i] * (hit * pHit + (1-hit) * pMiss))
    s = sum(q)
    for i in range (len(p)):
        q[i] = q[i]/s
    return q

print sense(p, Z)

def move(p, U):
    #Enter your code here

    return q

for k in range(len(measurements)):
    p = sense(p, measurements [k])
    print move (p, 1)
```

Answer 2 (Move Function)

```
def move(p, U):
    q= [ ] #Start with empty list
    for i in range(len(p)):
        #Go through all the elements in ''''''''p'''''
               #Construct  ''q''  element  by  element  by  accessi
corresponding ''p'' which is shifted by ''U''
        q.append(p[(i-U) % len (p)])
    return q

for k in range(len(measurements)):
p = sense(p, measurements [k])
print move (p, 1)
```

The (i-U) in the above function may be confusing at first.

If you are moving to the right, you may be tempted to use a plus sign. Don't! Instead of thinking of it as p shifting to a cell in q, think of this function as q grabbing from a cell in p.

For example, if we are *shifting* p one to the right, each cell in q has to grab its value from the cell in p that is one to the left.

Inaccurate Robot Motion: Localization Is Hard

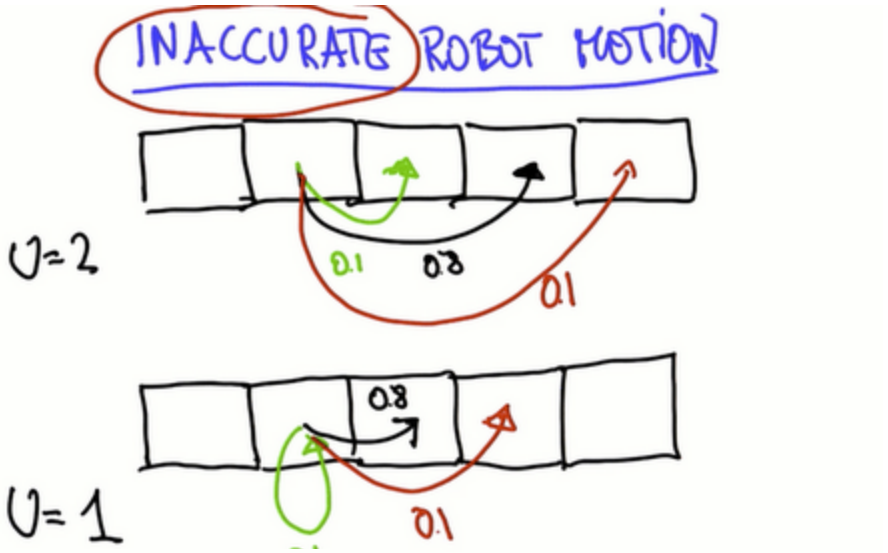
Question 1 (Inexact Motion)

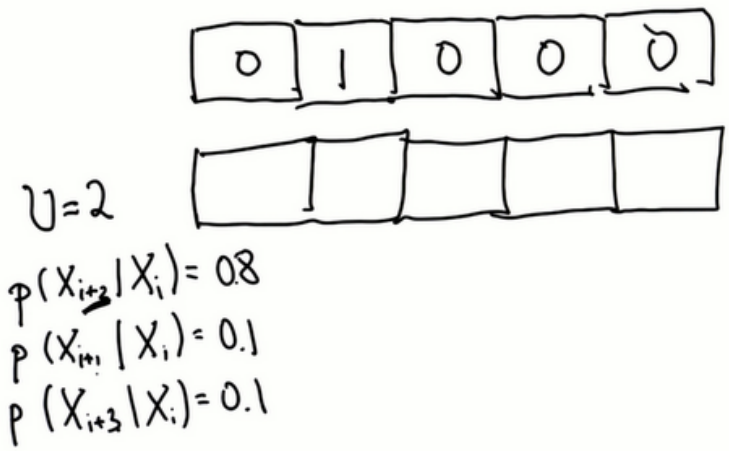
Assume the robot executes its action with high probability (0.8), correctly; with small probability (0.1) it will overshoot, and with small probability (again 0.1) it will undershoot.

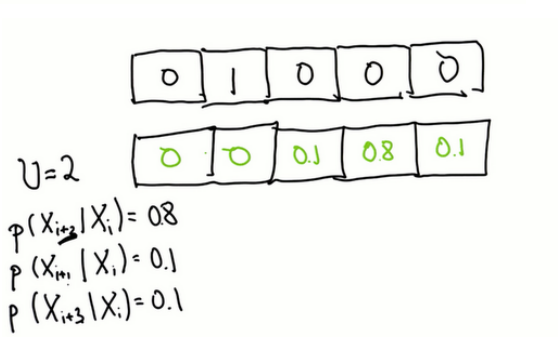
Note that this is a more accurate model of real robot motion.

Given a prior distribution, can you give the distribution after the motion?

Use this formula:





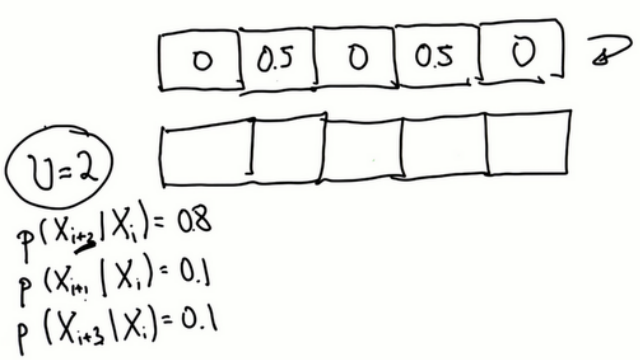


Answer 1 (Inexact Motion)

As expected, the probability distribution has shifted and spread out. Moving causes a loss of information.

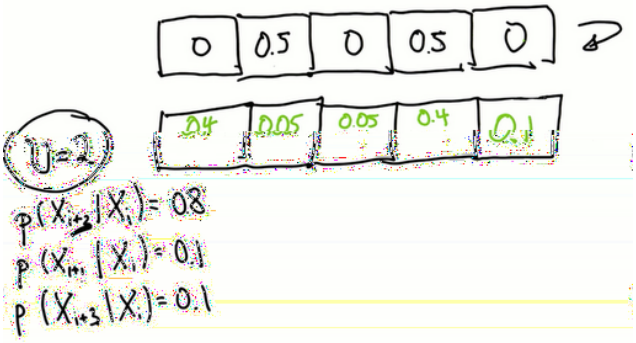
Question 2 (Inexact Motion)

This time, given that cells 2 and 4 have a value of 0.5, fill in the posterior distribution.



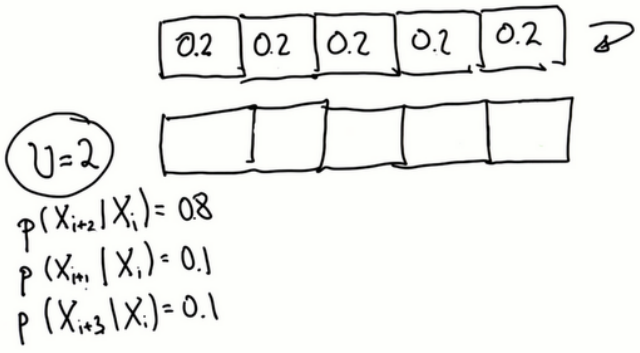
Answer 2 (Inexact Motion)

We answer this in the same way as in the previous question, but this time we have to take into account that cell 5 can be arrived at in 2 ways: by cell 4 undershooting or cell 2 overshooting.



Question 3 (Inexact Motion)

Given a uniform distribution, fill in the distribution after motion, using the formula.



Answer 3 (Inexact Motion)

We can answer this question in two ways:

1. As shown: explicitly calculate the possible ways of arriving at each cell. This takes quite a bit of calculation, but will give us the correct answer.
2. Realize that if we start with a uniform prior distribution (state of maximum confusion), and we then move, we MUST end up with

the uniform distribution. Since this problem starts with a state of maximum confusion and *then* moves (which, remember, never increases our knowledge of the system), we must remain in the maximally confused state.

Question 4 (Inexact Move Function)

Modify the move procedure to accommodate the added probabilities.

```
p = [0, 1, 0, 0, 0)
world = ['green', 'red', 'red', 'green', 'green']
measurements = ['red', 'green']
pHit = 0.6
pMiss = 0.2
#Add exact probability
pExact = 0.8
#Add overshoot probability
pOvershoot = 0.1
pUndershoot = 0.1

def move(p, U):
    q= []
    for i in range(len(p)):
        q.append(p[(i-U) % len (p)])
    return q
```

Answer 4 (Inexact Move Function)

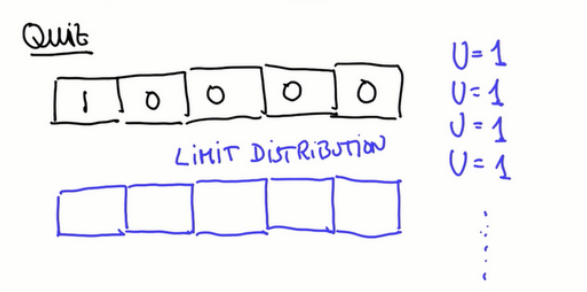
```
def move(p, U):
    #Introduce auxiliary variable s
    q= []
    for i in range(len(p)):
        s = pExact * p[(i-U) % len(p)]
        s = s + pOvershoot * p[(i-U-1) % len(p)]
        s = s + pUndershoot * p[(i-U+1) % len(p)]
        q.append(s)
    return q

[0.0, 0.1, 0.8, 0.1, 0.0]
```

This function accommodates the possibility of undershooting or overshooting our intended move destination by going through each cell in p and appropriately distributing its probability over three cells in q (the cell it was intended to go to, distance U away, and the overshoot and undershoot cells).

Question 5 (Limit Distribution Quiz)

What happens if the robot never senses but executes the motion to the right forever. What will be the limit or stationary distribution in the end?

A distribution is stationary when it doesn't change over time. In the case of the moving robot, this corresponds to the probability distribution *before* a move being the same as the distribution *after* the move.

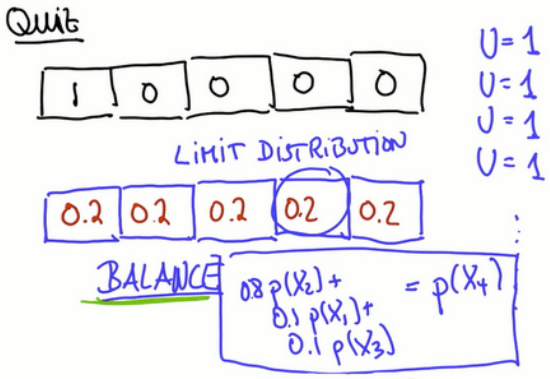


Answer 5 (Limit Distribution Quiz)

The fact that the result is a uniform distribution should not be surprising.

Remember Inexact Motion 3, where we saw that when we act on the uniform distribution with the function move, we get back the uniform distribution. Moving doesn't affect the uniform distribution, and this is exactly the definition of a stationary state!

We can also solve for the probability of each cell by realizing that each cell is the possible destination for three other cells. This is the method used in the lecture and shown below. Of course, regardless of our method, we get the same answer: the uniform distribution.



Question 6 (Move Twice)

Write code that makes the robot move twice, starting with the initial distribution:

```
p = [0, 1, 0, 0, 0]
```

Answer 6 (Move Twice)

```
p = move(p, 1)
p = move(p, 1)

print p

[0.01, 0.01, 0.16, 0.66, 0.16]
```

The result is a vector where 0.66 is the largest value and not 0.8 anymore. This is expected: two moves has flattened and broadened our distribution.

Question 7 (Move 1000)

Write a piece of code that moves 1,000 steps and give you the final distribution.

Answer 7 (Move 1000)

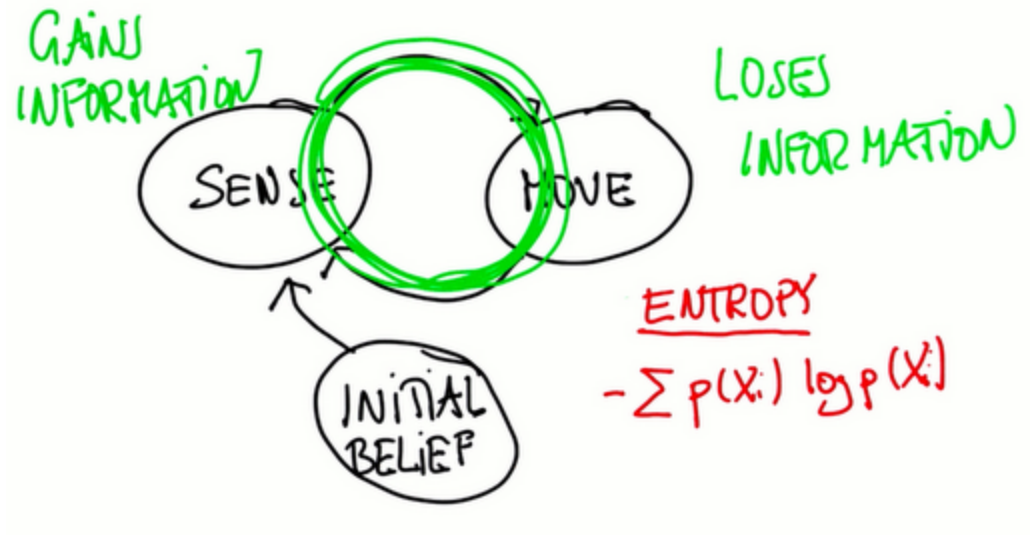
```
#Write a loop for 1,000 steps
for k in range(1000):
    p = move(p, 1)
    print p
    #Final distribution is 0.2 in each case as expected
```

After 1000 moves, we have lost essentially all information about the robot's location. We need to be continuously sensing if we want to know the robot's location.

Robot Sense and Movement

Localization, specifically the Monte Carlo localization method that we are using here, is nothing but the repetition of *sensing* and *moving*. There is an initial belief that is tossed into the loop. If you sense first it comes to the left side. Localization cycles through move and sense. Every time it moves it loses information and every time it senses it gains information. Entropy is a measure of the information that a distribution has.

Entropy and information entropy are both fascinating topics. Feel free to read more about them!



Question 1 (Sense and Move)

Given the motions one and one, which means the robot moves right and right again:

```
motions = [1, 1]
```

Compute the posterior distribution if the robot first senses red, then moves right by one, then senses green, then moves right again. Start with a uniform prior distribution:

```
p = [0.2, 0.2, 0.2, 0.2, 0.2]
```

Answer 1 (Sense and Move)

```
for k in range(len(measurements)):
    p = sense(p, measurements[k])
    p = move(p, motions[k])
    print p
```

The robot likely starts in grid cell 3, which is the right-most of the two red cells.

Question 2 (Sense and Move)

Can you modify the previous question so that the robot senses red twice. What do you think will be the most likely cell the robot will reside in?

How will this probability compare to that of the previous question?

Answer 2 (Sense and Move)

The most likely cell is cell 4 (not cell 3—don't forget we move after each sense).

This program is the essence of Google's self-driving car's Monte Carlo localization approach.

Localization Summary (Localization Summary)

Localization involves a robot continuously updating its belief about its location over all possible locations. Stated mathematically, we could say the robot is constantly updating its *probability distribution* over the *sample space*. For a real self-driving car, this means that all locations on the road are assigned a probability. If the AI is doing its job properly, this *probability distribution* should have two qualities:

1. It should have one sharp peak. This indicates that the car has a very specific belief about its current location.
2. The peak should be correct! If this weren't true, the car would believe—very strongly—that it was in the wrong location. This would be bad for Stanley.

Our Monte Carlo localization procedure can be written as a series of steps:

1. Start with a belief set about our current location.
2. Multiply this distribution by the results of our sense measurement.
3. Normalize the resulting distribution.
4. Move by performing a convolution. This sounds more complicated than it really is: this step really involved multiplication and addition to obtain the new probabilities at each location.

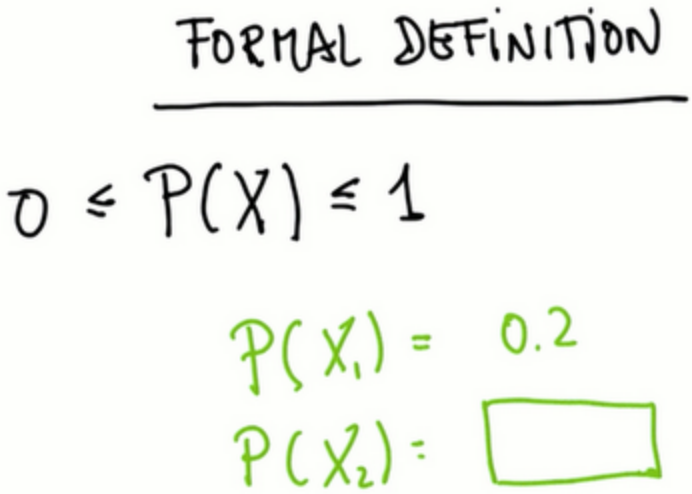
Keep in mind that step 2 always increases our knowledge and step 4 always decreases our knowledge about our location.

Extra Information: You may be curious about how we can use individual cells to represent a road. After all, a road isn't neatly divided into cells for us to jump between: mathematically, we would say the road is not *discrete* but rather *continuous*. Initially, this seems like a problem. Fortunately, whenever we have a situation where we don't need exact precision—and remember that we only need 2-10 cm of precision for our car—we can chop a continuous distribution into pieces and make those pieces as small as we need.

In the next unit we will discuss Kalman Filters, which use continuous probability distributions to describe beliefs.

Question 1 (Formal Definition of Probability)

Remember that a probability of zero means an event is impossible, while a probability of one means it is certain. Therefore, we know all probabilities must satisfy $0 \leq P(x) \leq 1$. Let's assume that X_1 and X_2 are complementary events: either one or the other must happen. If $P(X_1) = 0.2$, what is $P(X_2)$?



Answer 1 (Formal Definition of Probability)

The answer is 0.8.

Question 2 (Formal Definition of Probability)

If $P(X_1) = 0$, what is $P(X_2)$?

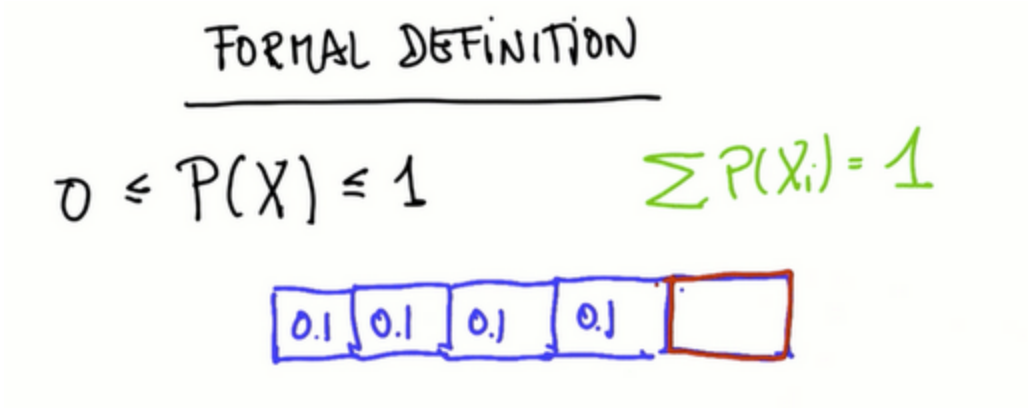
Answer 2 (Formal Definition of Probability)

$P(X_2) = 1$.

In both of these problems we used the fact that the sum of all probabilities must add to one. Probability distributions must be normalized.

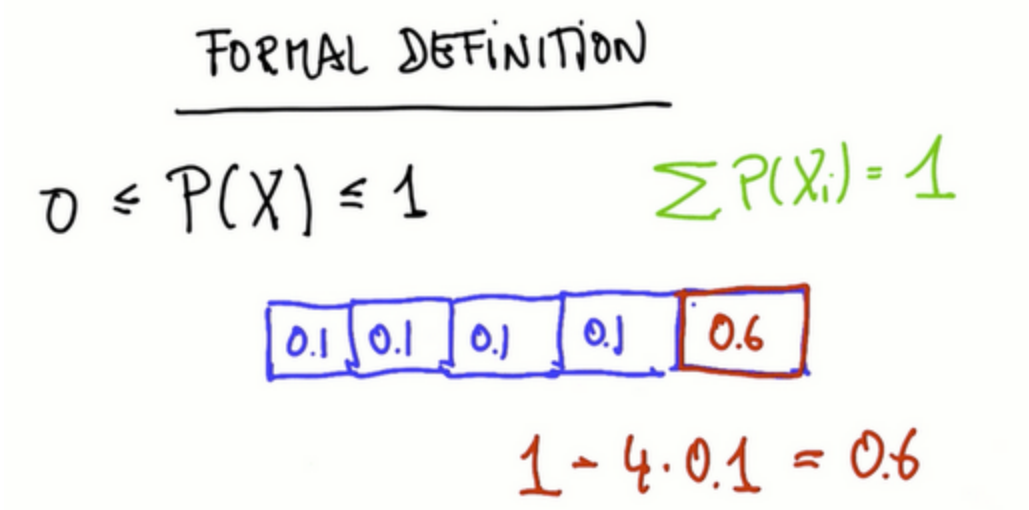
Question 3 (Formal Definition of Probability)

Given:



Fill in the value of the fifth grid cell.

Answer 3 (Formal Definition of Probability)



Bayes' Rule

Bayes' Rule is a tool for updating a probability distribution after making a measurement.

- x = grid cell
- Z = measurement

The equation for Bayes' Rule looks like this:

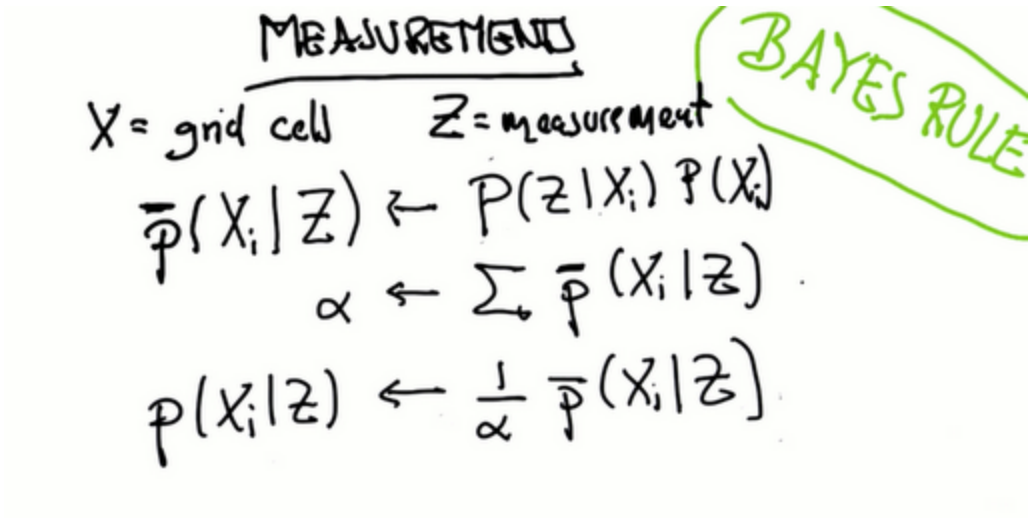
$$P(x|Z) = \frac{P(Z|x)\,P(x)}{P(Z)}$$

The left hand side of this equation should be read as "The probability of X after observing Z." In probability, we often want to update our probability distribution after making a measurement. Sometimes, this isn't easy to do directly.

Bayes' Rule tells us that this probability (the left-hand side of the equation) is equal to another probability (the right-hand side of the equation), which is often easier to calculate. In those situations, we use Bayes' Rule to rephrase the problem in a way that is solvable.

To use Bayes' Rule, we first calculate the non-normalized probability distribution, which is given by *P(Z|X)P(X)*, and then divide that by the total probability of making measurement Z, P(Z), which is called the normalizer.

This may seem a little confusing at first. If you are still a little unsure about Bayes' Rule, continue on to the next example to see how we put it into practice.



Cancer Test Question

Suppose there exists a certain kind of cancer that is very rare:

- *P(C) = 0.001* = Probability of having cancer
- *P(¬C) = 0.999* = Probability of not having cancer

Suppose we have a test that is pretty good at identifying cancer in patients with the disease (it does so with 0.8 probability), but occasionally (with probability 0.1) misdiagnoses cancer in a healthy patient:

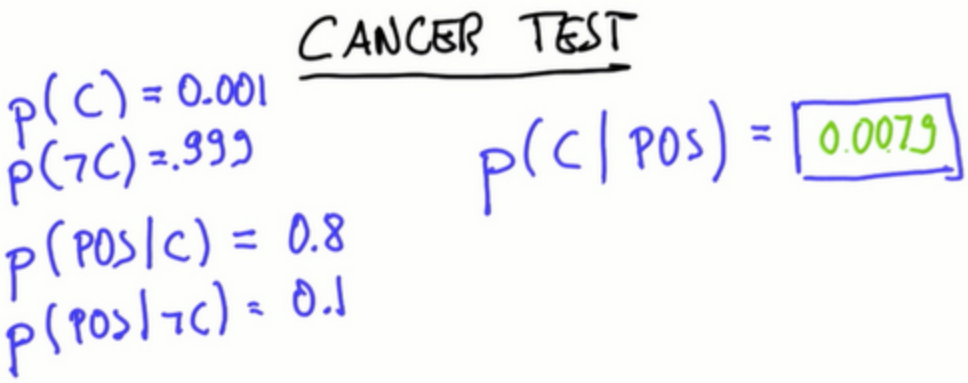
- P(POS|C) = 0.8 = Probability of a positive test if patient has cancer

- *P(POS|¬C) = 0.1* = Probability of a positive test in a cancer-free patient (a “false positive")

Using this information, can you compute the probability of having cancer given that you receive a positive test?

- *P(C|POS) = ?*

Answer: (Cancer Test)



Only 0.79 out of 100 who test positive have cancer, even though they have positive cancer test results! (This math assumes there was no outside reason to test the individuals for cancer: they were randomly selected to be tested.)

You can apply the same mechanics as before to get this result. The non-normalized result of Bayes’ Rule is the product of the prior probability, P(C), multiplied by the probability of a positive test, *P(POS|C)*:

- *P(C|POS)=*

Plug in the probabilities:

- *P(C|POS) = 0.001 * 0.8 = 0.0008*

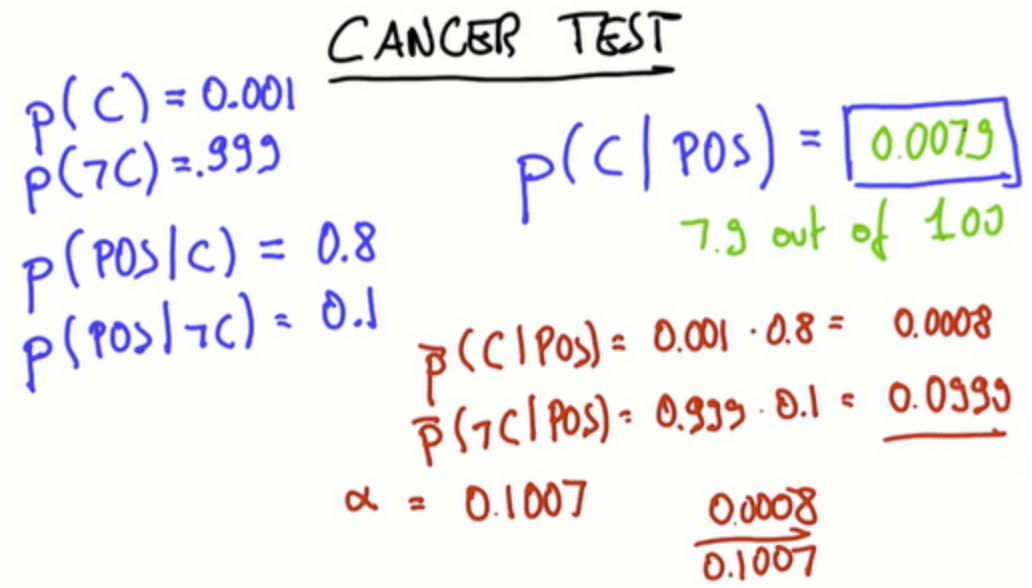
Non-normalized probability for the opposite event, given a positive test:

- *P(¬C|POS) = 0.999 * 0.1 = 0.0999*

Normalized (the sum of the the probability of having cancer after a positive test and not having cancer after a positive test):

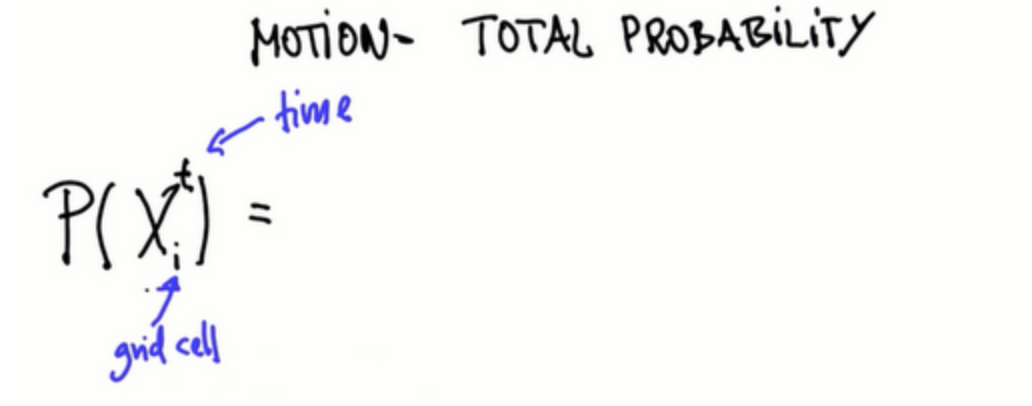
- *P(C|POS) + P(¬C|POS) = 0.1007*

Dividing the non-normalized probability, 0.0008, by the normalized probability, 0.1007, gives us the answer: 0.0079.



Theorem of Total Probability

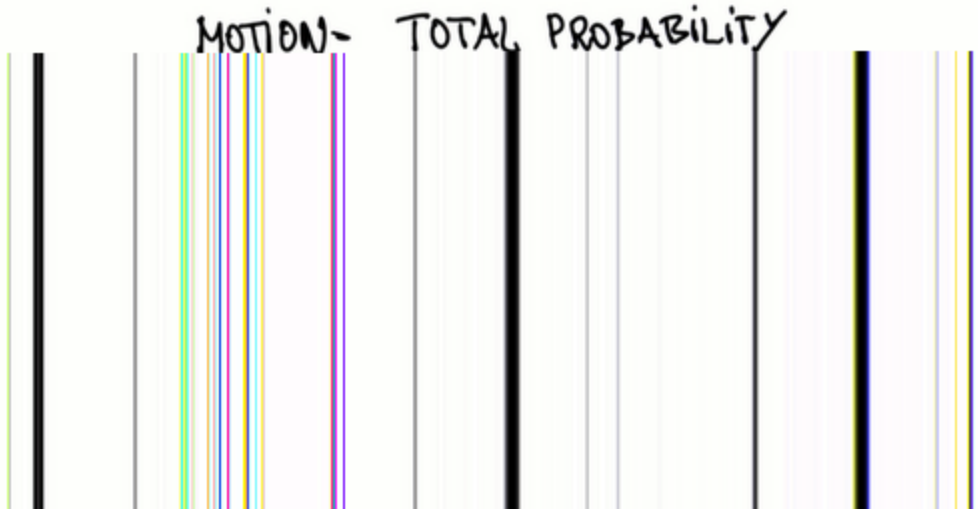
Now let's look at motion, which will turn out to be something we will call total probability. Hopefully you remember caring about a grid cell "xi" and we asked what is the chance of being in xi after robot motion. Now, we will use a time index to indicate after and before motion:



Compute this by looking at all the grid cells the robot could have come from one time step earlier (at time t-1), and index those cells with the letter j, which in our example ranges from 1 to 5. For each of those cells, look at the prior probability $P(X_j^{t-1})$, and multiply it with the probability of moving from cell X_j to cell X_i, $P(X_i|X_j)$. This gives us the probability that we will be in cell X_i at time t:

-

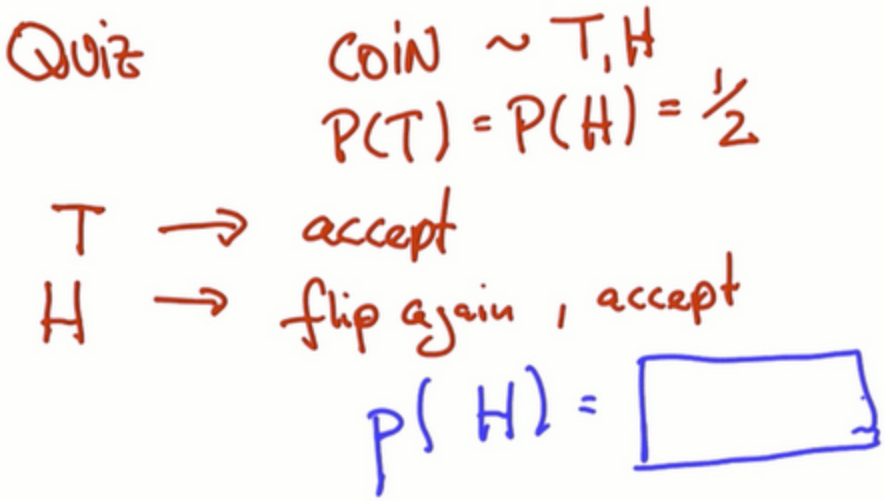
You can see the correspondence of *A* as a place *i* of time *t* and all the different *Bs* as the possible prior locations. This is often called the Theorem of Total Probability.



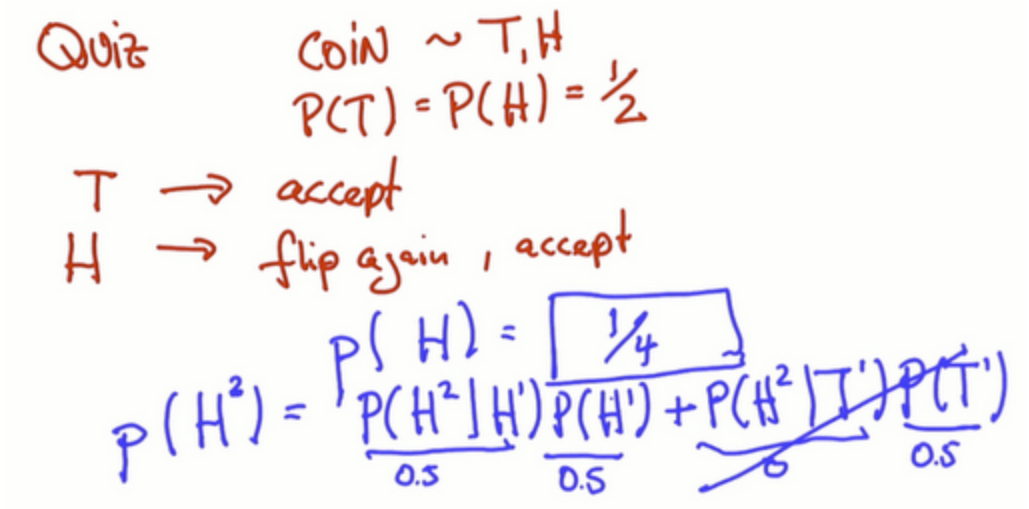
Question 1 (Coin Flip Quiz)

Coin Toss: The probability of tossing a fair coin, heads (H) or tails (T) is: *P(T) = P(H) = 0.5.*

If the coin comes up tails, you stop and accept the result; but if it comes up heads, you flip it again and then accept the result. What is the probability that the final result is heads?



Answer (Coin Flip Quiz)

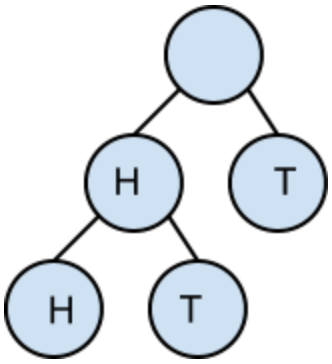


The probability of throwing heads in step two, $P(H^2)$, depends upon having thrown heads in step one, $P(H^1)$, which we can write as a condition, $P(H^2|H^1)P(H^1)$. Add the probability of throwing heads in step two with the condition, $p(H^2|T^1)P(T^1)$, of throwing tails in step one, multiplied by the probability of throwing tails in step one, $P(T^1)$. This can be written as

- $P(H^2)=P(H^2|H^1)P(H^1)+P(H^2|T^1P(T^1)$

The last term of this equation, $P(H^2|T^1P(T^1)$, is the product of the probability of heads given a tail on the first throw with the probability of tails on the first throw. Since we are told that we stop flipping when we get a tails on the first throw,$P(H^2|T^1) = 0$, and we can ignore this term. Our equation becomes $P(H^2) = P(H^2|H^1)P(H^1)$, which is $\frac{1}{2} = \frac{1}{4} * \frac{1}{4}$.

Note that the superscripts next to H and T in this problem indicate whether we are talking about the first or second toss. They are not exponents.



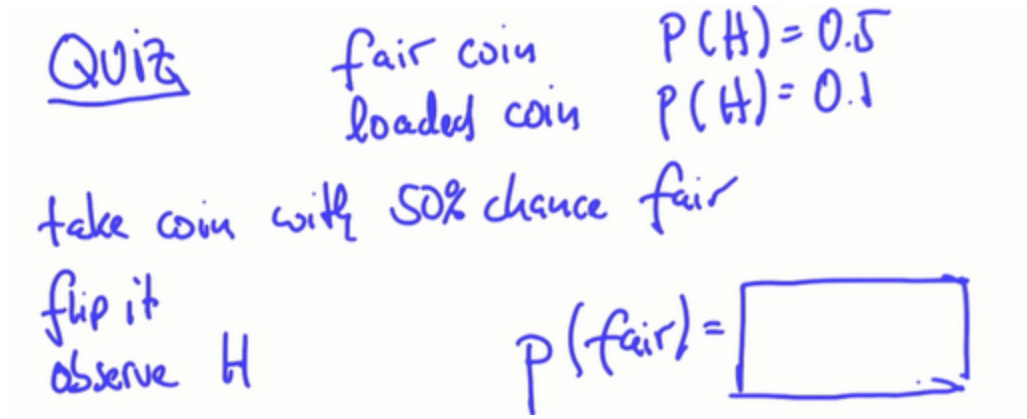
We can also approach this problem as a probability tree, where the probability of moving down any branch is 1/2. We can see that the only way to arrive at heads on the second toss is to proceed down the heads path twice.

Question 2 (Two Coin Quiz)

Say you have two coins, one of which is fair and the other loaded. The probability of flipping heads for each coin is shown below.

- fair coin: P(H) = 0.5
- loaded coin: P(H) = 0.1

There is a 50% chance of flipping either the fair or the loaded coin. If you throw heads, what is the probability that the coin you flipped is fair?



Answer 2 (Two Coin Quiz)

The question, what is the probability of throwing a fair coin given that you observe H, can be written as:

- *P(F|H) = ?*

Use Bayes' Rule because you are making observations. The non-normalized probability, which we will represent with a lower case p, of getting the fair coin is:

- *p(F|H) = P(H|F) P(F)*
- *p(F|H) = 0.5 * 0.5*

The non-normalized probability of NOT getting the fair coin, but getting the loaded coin is:

- *p(¬F|H) = P(H|¬F) P(¬F)*
- *p(¬F|H) = 0.1 * 0.5*

When you find the sum of these, the result is:

- *0.25 + 0.05 = 0.3*

Now, we divide our non-normalized probability by this sum to obtain our answer: $\frac{0.25}{0.3}$ = 0.833.

Conclusion (Conclusion)

This is what you've learned in this class:

- Localization
- Monte Carlo Localization
- Probabilities
- Bayes' Rule
- The Theorem of Total Probability

You are now able to make a robot localize, and you have an intuitive understanding of probabilistic methods called *Filters.*Next class we will learn about:

- Kalman Filters