AI for Robotics 1: Localisation

David Peebles

14th February 2016

Sensing

This lesson introduces *Monte Carlo Localisation* using discrete spaces (which can be represented by a histogram). This method (unlike the Kalman Filters in Lesson 2) can be used with multi-modal distributions. This is the problem of finding where you are in an environment in relation to a representation (or map) of the environment. The basic idea is that the robot has a *belief function* (a probability distribution) that represents the probability of it being in every location in the environment.

If the belief function is uniform, then the robot has no idea where it is (each location is equally likely). If, through *sensing* the environment, the robot identifies a landmark, then it can update the *prior* probability distribution to create a *posterior* probability distribution that reflects this.

Here is an array, p of five cells x_i with equal probability.

```
n = 5                      # number of cells
x = round(1.0/n,3)         # equal probability over n
p = [x] * n                # array of n x values
p
```

```
0.2   0.2   0.2   0.2   0.2
```

Probability after sensing

Given this array of five cells x_1 to x_5 which can be either red, r or green, g, (g, r, r, g, g), the robot can sense the colour of one of the cells (Z). How should this information be used to update the robot's belief about where it is (i.e., how do we create the posterior distribution?), $p(x_i|Z)$.

This can be done using a simple multiplicative rule. If the cell is the same colour as that sensed, multiply it by the value **0.6**. If it is a different colour to the one sensed, multiply it by the value **0.2**. To create a probability distribution from the new array, we have to normalise the values by dividing them all by their sum. This ensures that the values sum to 1.

```
pHit = 0.6
pMiss = 0.2

for i in range(len(p)):
    if i == 1 or i == 2:
        p[i] = p[i] * pHit
    else:
        p[i] = p[i] * pMiss

sm = sum(p)
[round(j / sm,3) for j in p]
```

```
0.111   0.333   0.333   0.111   0.111
```

A sense function

We now extend this by creating a *world* array representing the colours of the cells and the variable Z representing the sensed colour. We then define a function that takes the world and the sensed colour as arguments and returns the normalised posterior distribution.

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

def sense(p, Z):
    q = []
    for i in range(len(p)):
        if world[i] == Z:
            q.append(p[i] * pHit)
        else:
            q.append(p[i] * pMiss)
    sumprob = sum(q)
    return [round(j / sumprob,3) for j in q]

sense(p,Z)
```

```
0.111   0.333   0.333   0.111   0.111
```

We can test that code with the measurement finding green instead of red...

```
Z = 'green'
sense(p,Z)
```

```
0.273   0.091   0.091   0.273   0.273
```

Multiple measurements

If we want to carry out multiple sequential measurements, we can just create a list of measurements and run through them, updating p each time...

```
measurements = ['red', 'green']
for m in measurements:
    p = sense(p, m)

p
```

```
0.2   0.2   0.2   0.2   0.2
```

Robot motion

For **exact** motion in an array of cells over which a probability distribution has been defined, the distribution is simply shifted the number of cells in the direction of movement. We assume that the array is circular so that moving past the end of the array lands you at the other end. The **move** function defined below takes the array p and the number of cells moved (positive values indicate movement to the right, negative to the left) and returns a new array with the updated distribution.

The function needs some explanation. The new array is created by taking each cell number in p, subtracting U from it (this is crucial because it looks to where the cell has originated from), and then getting the modulo from dividing that value by the length of the array. This last operation creates the circular effect.

```
p = [0, 1, 0, 0, 0]
def move(p, U):
    q = []
    for i in range(len(p)):
        q.append(p[(i-U)%len(p)])
    return q

move(p, 1)
```

```
0   0   1   0   0
```

For **inexact** robot motion, the situation is different. Instead of just shifting the probability distribution by the number of cells, we have to update the probability distribution by multiplying the cell values by the probabilities that the robot has moved to that cell. The robot may only be able to move to its target cell with a certain probability and may overshoot and undershoot by a certain probability.

The **inexactMove** function below incorporates this uncertainty by setting the probability of moving to the desired cell to .8 and the probability of overshooting and undershooting to .1.

```
p = [0, 1, 0, 0, 0]
pExact = 0.8
pOvershoot = 0.1
pUndershoot = 0.1

def inexactMove(p, U):
    q = []
    Ua = U-1
    Ub = U+1
    for i in range(len(p)):
        tot = (pExact * p[(i-U)%len(p)]) \
              + (pOvershoot * p[(i-Ub)%len(p)]) \
              + (pUndershoot * p[(i-Ua)%len(p)])
        q.append(tot)
    return q

inexactMove(p, 1)
```

```
0.0   0.1   0.8   0.1   0.0
```

We can now run this inexact move function twice…

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

def moveTimes(p, n):
    for x in xrange(n):
        p = list(inexactMove(p, 1))
    return [round(elem, 3) for elem in p]

moveTimes(p, 2)
```

```
0.01   0.01   0.16   0.66   0.16
```

Or 100 times…

```
p = [0, 1, 0, 0, 0]
moveTimes(p, 100)
```

```
0.2   0.2   0.2   0.2   0.2
```

This last example shows that with inexact movement, after a certain number of moves, the robot will have no idea where it is.

Sensing and moving combined

So the basic idea is that the robot has a prior belief about its location which is then updated by repeatedly looping through the **sense** and **move** procedures. Each time it senses the environment, it gains information about its location (i.e., the probability distribution is a little bit more defined and 'spikier') while every time it moves, it loses in information about its location (i.e., the probability distribution is a little bit flatter).

The code below includes a new **motions** variable that represents a list of movements and implements a sequence of two sense and two move actions.

```
p=[0.2, 0.2, 0.2, 0.2, 0.2]
measurements = ['red', 'green']
motions = [1,1]

for x in xrange(2):
    p = sense(p, measurements[x])
    p = inexactMove(p, motions[x])

[round(elem, 3) for elem in p]
```

```
0.211   0.152   0.081   0.168   0.387
```

We reset the probabilities and repeat this with two different measurements...

```
p=[0.2, 0.2, 0.2, 0.2, 0.2]
measurements = ['red', 'red']

for x in xrange(2):
    p = sense(p, measurements[x])
    p = inexactMove(p, motions[x])

[round(elem, 3) for elem in p]
```

```
0.079   0.075   0.225   0.433   0.189
```

Bayes rule

- The robot's belief about its location is a probability distribution over discrete locations.
- Sensing is a product followed by a normalisation. Motion is a *convolution*. This means that for each location after the motion, we attempt to 'reverse engineer' the situation (i.e., guessed where the world might have come from) and then added the corresponding probabilities.
- The measurements use *Bayes rule* shown in Equation 4. If x is a grid cell and Z is a measurement, then the measurement update seeks to find the belief over the location after the measurement, $p(x|Z)$. In this equation, $p(x)$ is the prior probability while $p(Z|x)$ is the chances of seeing a red or green tile for every possible location. The numerator gives the non-normalised posterior distribution while the denominator, $p(Z)$ is the normalising term which is the sum of the resulting probabilities.

$$p(x|Z) = \frac{p(Z|x) \cdot p(x)}{p(Z)} \tag{1}$$

If we think of the relationship for each cell $p(x_i|Z) = p(Z|x_i) \cdot p(x_i)$, the $p(Z|x_i)$ was 0.6 if the measurement corresponded to a correct colour and 0.2 if the measurement corresponded to an incorrect colour.

A Bayes rule example from a different domain

The probability of having a particular form of cancer, $p(c)$ is 0.001 while that of not having the cancer, $p(\neg c)$ is 0.999. There is a test for this cancer. The probability of the test being positive if you have the cancer, $p(pos|c)$ is 0.8 while the probability of a false positive, $p(pos|\neg c)$ is 0.1. Calculate the probability of having cancer given you have a positive test, $p(c|pos)$.

The prior, $p(c)$ is 0.001 and the measurement update, $p(pos|c)$ is 0.8. So the non-normalised posterior distribution, produced by $p(pos|c) \cdot p(c)$ is $0.8 \times 0.001 = 0.0008$. To normalise, we have to calculate the sum of all the outcomes. For this example, it is the positive case above (0.0008) plus the negative case, $p(pos|\neg c) \cdot p(\neg c)$ is $0.1 \times 0.999 = 0.0999$. The sum of 0.0999 and 0.0008 is 0.1007. Therefore when we divide the numerator (0.0008) by this denominator (0.0999) the final result is **0.0079**. This result means that if you get a positive result in the test, you have a .79% (.79 out of a 100) chance of having the cancer.

Motion – Total probability

The probability of being in a grid cell at time t, $p(X_i^t)$ is computed according to Equation 2

$$p(X_i^t) = \sum_j p(X_j^{t-1}) \cdot p(X_i|X_j) \tag{2}$$

This equation states that the probability of grid cell X_i at time t is computed from the prior probabilities of all of the cells j which could lead to X_i at time $t-1$ multiplied by the probability that the motion command would take us to X_i (e.g., 0.8 to the target cell and 0.1 for overshoot and undershoot), denoted by $p(X_i|X_j)$.

This is sometimes represented as the *theorem of total probability*, shown in Equation 3 and the right hand side of the equation is often called a *convolution*.

$$p(A) = \sum_B p(A|B) \cdot p(B) \tag{3}$$

A total probability quiz

Take a fair coin and flip it. If it comes up tails just accept the result but if it comes up heads flip it again and then accept the result. What is the total probability that the result is heads?

The probability of heads in step 2, $p(H^2) = p(H^2|H^1) \cdot p(H^1) + p(H^2|T^1) \cdot p(T^1)$.

The probability of $p(H^1)$ and $p(T^1)$ are both 0.5. However the probability of $p(H^2|T^1)$ is zero because if we get tails in step 1, we just accept the result. Therefore $p(H^2|T^1) \cdot p(T^1) = 0 \times 0.5 = 0$.

Therefore the probability of heads in step 2, $p(H^2) = p(H^2|H^1) \cdot p(H^1) = 0.5 \times 0.5 = 0.25$.

A Bayes rule quiz

You have two coins. One is fair with the probability of heads being 0.5, the second is loaded with the probability of heads being 0.1. Take one of the coins at random and flip it. You observe that it comes up heads. What is the probability that the coin you chose is the fair one?

This boils down to the question of what is the probability of the fair coin having observed heads $p(F|H)$. Applying Bayes rule…

$$p(F|H) = \frac{p(H|F) \cdot p(F)}{p(H)} \tag{4}$$

The probability of getting heads with the fair coin $p(H|F)$ is 0.5 while the probability of getting the fair coin in the first place $p(F)$ is also 0.5. So the numerator is $0.5 \times 0.5 = 0.25$. We then calculate the denominator by summing all outcomes. For this example, it is the fair coin case above (0.25) plus the loaded dice case, $p(H|\neg F) \cdot p(\neg F)$ is $0.1 \times 0.5 = 0.05$. The sum of 0.25 and 0.05 is 0.3. Therefore when we divide the numerator (0.25) by this denominator (0.3) the final result is **0.83**. This result means that if you get a heads outcome when you flip the coin, you have a 83% chance of it being the fair coin.