60019 Robotics 2021 Past Paper Solution

Note: this is not an official solution but I'm trying to make it as accurate as possible. If you find any typos or incorrect answers and explainations, please email me at xz1919@ic.ac.uk with the title <course_number>-<course_name>-<year>-past-paper-enquiries or directly message me through Whatsapp. If you need explaination or reviews on the provided answer, please contact me as well. :) A copy of the exam feedback is provided at the end of the document as well

---

1. a. Write a program in Lua pseudocode to give the robot the repetitive behaviour described in the question prompt.

```
function initCallback()
  --the current state can be either of: "forward_until_bump", "random_turn",
"turn_right", "wall_follow"
  currentState = "forward_until_bump"
  speedBase = 5.0
  angleBase = 1.0
  motorAngleTarget = 0.0
  bumpingDistance = 0.25
  wallFollowingTimer = 0.0
  wallFollowingDistance = 0.5
  proportionalGain = 1.0
end

function actuationCallback()
  distance = readSonar()
  v = 0.0
  omega = 0.0

  pos = sim.getRobotWheelDirection()

  if (currentState == "forward_until_bump") then
    v = speedBase
    omega = 0.0
    if (distance < bumpingDistance) then
      rand = math.random()
      if (rand < 0.8) then
        motorAngleTarget = math.rad(math.random() * 360)
        currentState = "random_turn"
      else
        motorAngleTarget = math.rad(pos - math.rad(90))
        currentState = "turn_right"
      end
    end
  elseif (currentState == "random_turn") then
    v = 0.0
    omega = angleBase
    if (pos >= motorAngleTarget) then
      currentState = "forward_until_bump"
    end
  elseif (currentState == "turn_right") then
    v = 0.0
    omega = angleBase
    if (pos >= motorAngleTarget) then
      setTurretAngle(math.rad(90))
      wallFollowingTimer = getTime()
      current_state = "wall_follow"
    end
  elseif (currentState == "wall_follow") then
    v = speedBase
    omega = proportionalGain * (distance - wallFollowingDistance)
    if (getTime() - wallFollowingTimer >= 10.0) then
      motorAngleTarget = math.rad(math.random() * 360)
      currentState = "random_turn"
    end
  end

  setVelocity(v, omega)
end
```

Comment on the code snippet:

There are lots of things to notice in this question. Below is a list of points the markers might care about.

- The positivity/negativity of the rotation is super important. In the first paragraph of the question prompt, it mentions that left direction is positive, so we need to stick to this throughout the entire code. There are 3 places we need to take care: when turning the robot to the right, when turning the turret to the left, and when we fine-tune the omega during wall-following using propotional feedback control
- Notice the degree to radian conversion in the code
- The variables `speedBase`, `angleBase`, `wallFollowingDistance`, and `bumpingDistance` can be set to any suitable values (using either fine-tuning or some specification). The variable `proportionalGain` should be tuned based on performance.
- It's necessary to have at least four states not only because the question prompt has suggested it, but because the description implies there are at least 4 states the robot can be in. This is a bit tricky since some people might think turning towards the wall and wall-following can be finished in one step, but in reality, these are two separate steps that both hold a stateful variable (e.g. turning towards the wall needs a `motorAngleTarget` and wall-following needs a `wallFollowingTimer`), hence they are two different states
- Since the question prompt did not mention this, we might need to state the assumption that the function `sim.getRobotWheelDirection()` exists and we are allowed to use it. If this is not allowed then we need another variable to record the approximate state of the variable wheel direction
- Notice the `getTime()` function needs to be written within the if-statement in the "turn_right" case. This is to make sure we start to time the wall-following right before we get to it.
- Notice the `setTurretAngle(math.rad(90))` technically needs to be tuned. The math.rad(90) here just assumes the robot will always turn the sonar sensor to the left. So that if you want to mention the fact that this needs to be tuned, just write `setTurretAngle(tuned_angle)` and have the `tuned_angle` in the init function
- Notice that according to the question prompt, the function `setTurretAngle()` will set the orientation (in radians) of the turret. Hence, we don't need a separate state to "track" the process of rotating the turret. e.g. calling it once is enough to set the direction of the turret to face left, unlike `setVelocity(v, omega)` where we are actually controlling the speed.
- Of course you don't need to follow this structure of handling the transitions between different states. In the code above, the transition is handled in the if-statements of old states and get to be executed in the if-statements of new states. You can also separate the transition and execution into two big if-statements. (e.g. the first if-statement detects whether the current state has finished and set variables used for the next state, and the second if-statement executes according to the state we are in, such as driving forward or turning. You might need a flag to indicate whether the step has finished)

b. Discuss briefly some of the possible shortcomings of the performance of the robot executing your program. How could the performance of this kind of behavioural control be improved, in particular if you have the freedom to change the behaviour rules of the program or if you can also change the design of the robot such as by adding additional sensors?

The problem is that the angle between robot's direction and the wall might be too large after "turn_right" step, such that the measurement of `distance` in the code in 1a would be much larger than the perpendicular distance between the robot and the wall. This will make the turning velocity `omega` a large value, which indicates turning left. (since left is the positive direction) However, the robot in this case is almost facing the wall so that we need to turn right, not left.

There are several things we can do:

- We can use better PID control strategy so that the "turn_right" step completes properly. e.g. tuning the turning speed and acceleration, etc.
- Before the "wall_follow" step, we can rotate the turret 360 and make measurements every 10 degree, for instance. We take the angle with the smallest sensor reading as the "left" direction and then rotate the robot accordingly.
- We can mount the sensor in front of the wheels so that the rotation and distance to the wall are coupled
- We can add multiple sensors and then take the angle with the smallest reading to the the direction of "left" (and rotate the robot similarly as the second method)

c. An alternative design for a robot vacuum cleaner has sensors and algorithms which enable it to perform SLAM (simultaneous localisation and mapping) as it operates. Explain briefly (using sketches if you need too) how such a robot could use this capability to clean a room more efficiently than the behaviour-based robot.

In a behaviour-based setting, the robot does not know it's location within the global map. Hence, the robot will not recognize which part of the floor has been cleaned, nor does it know its current location which may help guide the decision. In contrast, the SLAM method allows the robot to recognize the relative position while constructing the map of the environment, so that it can estimate its location and make informed decision on where to move next.

A possible sketch/way of demonstrating this would be if we have a living room with a large space in the middle, several furnitures, and four walls surrounded. Using the behavior-based method the large middle region might be ignored/not fully covered, while using the SLAM method can make sure the middle region be all covered effectively.

2. A cleaning robot is to operate in the building whose floorplan is shown in the picture, which is divided into three rooms: an office (O), lab (L), and corridor (C). The robot has a perfect prior map of the building.

   a. When the robot switches itself on one morning, it knows that it is inside the building, but is otherwise completely lost, and believes that any position and orientation within the building is equally likely. What would be a sensible values with which it should initialise discrete prior probabilities P(O), P(L) and P(C) that it is in each of the three rooms?

   A sensible set of prior probabilities would be according to the proportion of each room to the entire floor plan.

   $P(O) = \frac{8 \cdot 2.5}{100} = \frac{20}{100} = 0.2$

   $P(L) = \frac{8 \cdot 7.5}{100} = \frac{60}{100} = 0.6$

   $P(C) = \frac{2*10}{100} = \frac{20}{100} = 0.2$

b. The robot has a simple temperature sensor, which when used reports a binary temperature measurement of either ‘hot’ or ‘cold’. During previous experience, the robot made many measurements using this sensor from known positions in the building. It found that when it was in the office, the temperature measurement reported was ‘hot’ 80% of the time; in the lab it was ‘hot’ 50% of the time; and in the corridor it was ‘hot’ on only 20% of occasions. After being switched on, the robot makes one measurement of temperature, and the sensor reports ‘cold’. Calculate updated discrete probabilities that the robot is in each of the three rooms. (You should not need a calculator if you do the calculation properly.)

From the question prompt we know that, given $P(Hot|\text{room})$ and $P(Cold|\text{room})$ represents the probability of detecting as ‘hot’/‘cold’ in a certain room,

$P(Cold|O) = 1 - P(Hot|O) = 1 - 0.8 = 0.2$

$P(Cold|L) = 1 - P(Hot|L) = 1 - 0.5 = 0.5$

$P(Cold|C) = 1 - P(Hot|C) = 1 - 0.2 = 0.8$

Because the robot is reporting ‘cold’, using Bayes Rule we have the updated probabilities:

$P(O|Cold) = \frac{P(Cold|O)P(O)}{P(Cold)} = \frac{0.2 \cdot 0.2}{0.5} = 0.08$

$P(L|Cold) = \frac{P(Cold|L)P(L)}{P(Cold)} = \frac{0.5 \cdot 0.6}{0.5} = 0.6$

$P(C|Cold) = \frac{P(Cold|C)P(C)}{P(Cold)} = \frac{0.8 \cdot 0.2}{0.5} = 0.32$

The fact that $P(Cold)$ equals to 0.5 can be derived either by normalize the numerator part of the probability OR directly calcuated using the Partition Rule of probability:

$$
\begin{aligned}
P(Cold) &= \sum_i P(Cold|\text{room}_i)P(\text{room}_i) \\
&= P(Cold|O)P(O) + P(Cold|L)P(L) + P(Cold|C)P(C) \\
&= 0.2 \cdot 0.2 + 0.5 \cdot 0.6 + 0.8 \cdot 0.2 \\
&= 0.5
\end{aligned}
$$

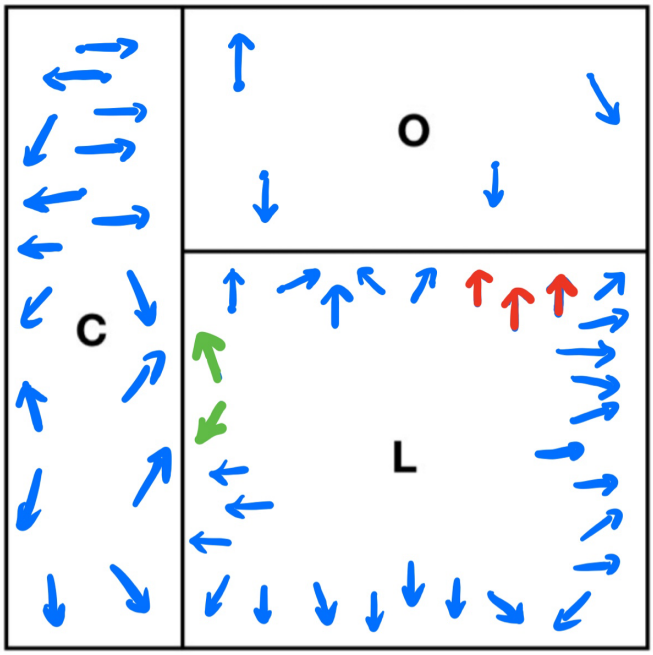
c. The robot needs a more precise estimate of its position, so after making the temperature measurement it initialises a particle filter to perform Monte Carlo Localisation (MCL) in a global localisation style. Describe briefly but precisely how the robot should initialise the parameters of 1000 particles to sensibly represent its state of knowledge after the temperature measurement.

For a global localization problem, we should randomly sample locations on the map, but with consideration of the prior knowledge after the temperature measurement.

We can divide the 1000 particles according to the updated probabilities of being in each room: we use 80 points to sample in the office area, 600 points to sample in the lab area, and 320 points to sample in the corridor. Then, for each room, we use a random number generator to generate random particles within that room.

Alternatively, you can directly distributes points according to the prior probabilities, e.g. you can first generate random number according to the prior probability to determine the particle’s location, then randomly drop it in the corresponding room.

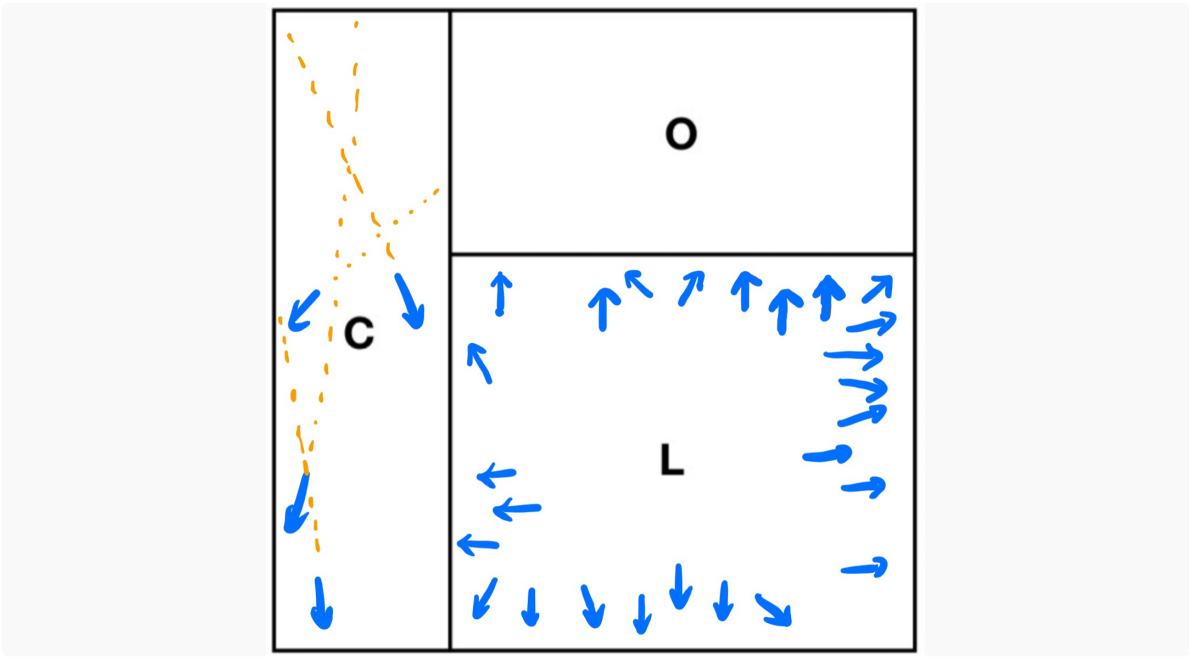
d. The robot has a horizontal forward-looking depth sensor mounted above its centre. The sonar has a standard deviation in its measurements of about 0.1m. Without moving, it uses this sensor one time, and receives a measurement of 1.0m. Assume that the rooms are empty of furniture or obstacles, but that the sensor will make a good measurement of any wall its beam hits. It uses this measurement to calculate likelihoods for all of the particles, then performs the standard normalisation and resampling steps of MCL. Sketch a copy of the diagram of the building, and on it draw 50 points with small arrows to visualise a typical set of particles which are likely to remain after resampling. (For each particle, the point will represent its (x, y) position and the arrow its θ orientation.)



Note that bcause the question doesn't specify, one should state clearly the assumption of which prior information to follow. In my case, I choose to follow the prior we generated in c. You can also choose to follow the prior probability in b. Either way, you should distribute the number of points according to the prior probability. (In the graph above, I have $0.08 \cdot 50 = 4$ points in the office, $0.6 \cdot 50 = 30$ points in the lab, and $0.32 \cdot 50 = 16$ points in the corridor)

The marking scheme might care about whether or not you include the standard deviation of 0.1m (arrows in red) AND whether or not you have different directions (arrows in green) because a robot might measure non-perpendicular distances.

e. The robot has a second identical depth sensor mounted very close to the first one, but this one points horizontally backwards. Without moving, it now uses this sensor one time and obtains a measurement of 5.0m, then again performs the likelihood calculation, normalisation and resampling steps of MCL to update the particle distribution. Make another sketch of the building and copy over any of the particles from your previous diagram which are still likely to survive after this update.



You should eliminate all the points whose tail distance to any wall is (at least visually) smaller than 5.0m. You can add a sentence or two to address the standard deviation as well.

f. If a point estimate of the position of the robot is needed to guide navigation, explain why taking the mean particle location is likely to give a poor estimate during the early stages of global localisation. Suggest an alternative which could work better.

In the early stage, the point cloud of estimation is distributed to different regions in the map, which might include significant outliers against the robot's true position. When taking the average, these extreme outliers will affect the robot position estimation. For example, there could be several cluster of point clouds that are way off from the true location of the robot, which results in an estimate that is located in the middle of those clusters.

A better way to do this is to take ***weighted*** average of those points to be the true location. In this way, the outliers will have less impact because they will have lower probabiltiy after sensor measurement updates.

Even better (and I think this is what the markers want to see): we should use clustering algorithms to classify each point into a cluster, and then calculate the mean probability of each cluster. We can just use the cluster with the highest probability to represent the real location of the robot.