vTopics not covered this year in comparison to previous years: Planning algorithms, tuning PID controllers, SLAM algorithm details (general overview is expected)

Q1)

(a)

|  |
| --- |
| def StoreSignature():  signature = []  for i in range(360): ￼  setSonarAngle(i - 179)  measurements = []  for i in range(5):  measurements.append(getSonarDepth())  medianMeasurement = median(measurements)  signature.append(medianMeasurement)  return signature |
|  |

(b)

|  |
| --- |
| # The bit in the notes about Location Learning def RecognisePlace(SavedSignatures):  # We first take a current reading  Signature = StoreSignature()  # We get the differences between our current reading and all saved readings  differences = []  for sig in SavedSignatures:  diffForThisSignature = 0  for i in range(360):  diffForThisSignature += (Signature[i] - sig[i])\*\*2  differences.append(diffForThisSignature)  # If even the smallest difference > ThresholdWeSet, we are at an unrecognised place  if min(differences) > THRESHOLD: # Some threshold we set  return -1  # Else, we are at the location with the lowest difference (The most like our current read)  return differences.index(min(differences)) |

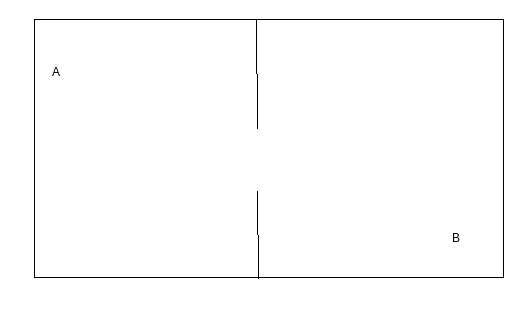
(c)  
# As mentioned, np.histogram(data) if allowed to use numpy

def ConvertToHistogram(Signature):

# As depth bin quantisation = 5, total bins = 200 / 5 = 40  
 bins = [0] \* 40  
 for reading in Signature:

bins[reading // 5] += 1  
 return bins

(d)



\_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_

| |

| |

| a b |

| |

| \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ \_\_ |

(e)

It could track its location based on its starting position and try to calculate how far it has moved since it started. In this scenario the robot would be able to use this information to compare these two points A, B compared to where it started and would easily consider them as separate points.

Another method would be whenever it encounters a scenario in which a signature could match two locations, track both of those positions and continue. It would then localise based on two maps and after more localisation hopefully one of the two robot copies would have better recognition based on where it is.  
--------------------------------------------------------------------------------------------------

(e)  
**One method (Uses continuous localisation)**  
ASSUME ambiguous Locations are within 199cm of at least one obstacle.  
We revert to an MCL-based approach. For Z number of ambiguous locations, generate X1...XN/Z particles at Location1, X(N/Z+1)...X(2N/Z) particles at Location 2, and so on.  
W1….WZ = 1/N  
For each ambiguous Location:  
 1. Scan 360degrees  
 2. Calculate P(z|m)  
 Where

z is the smallest sonar depth recorded  
 m is the expected sonar depth, or the distance between Location and the closest  
 Obstacle point to it. (We can derive this from both the depth histogram and map)  
 3. Update all weights belonging to Location, wi\_new = wi \* P(z|m)  
 4. Normalize weights  
 5. Resample  
After a few repeated iterations(You can even test other things like nearest corner, etc), eventually the particles belonging to unlikely ambiguous locations will die out, leaving behind the likely Location.   
  
**Another method (Does not use continuous localisation)**  
When learning a new area, the robot remembers both the Signature as well as the Depth Histogram. If the Depth Histogram produces an ambiguous solution, conduct RecognisePlace(), but only on the Signatures of locations that were in the ambiguous set. This will return the most likely Location

Q2

(a)

|  |
| --- |
| SIGMA= 2 K = 0.1  def GetLikelihood(particle, z, alpha, walls):  prob = 1.0  for i in range(20):  measurement = z[i]  angle = alpha[i]  dists = []  for wall in walls:  (Ax, Ay, Bx, By) = wall  (x, y, theta) = particle  theta = theta + angle   top = ((By - Ay)\*(Ax-x))-((Bx-Ax)\*(Ay-y))  bottom = ((By-Ay)\*math.cos(theta))-((Bx-Ax)\*math.sin(theta))   if bottom == 0:  continue   m = top / bottom  if m < 0:  continue  poi\_x = x + m \* math.cos(theta)  poi\_y = y + m \* math.sin(theta)   wall\_min\_x = min(Ax,Bx)  wall\_max\_x = max(Ax,Bx)  wall\_min\_y = min(Ay,By)  wall\_max\_y = max(Ay,By)    if wall\_min\_x <= poi\_x and  wall\_max\_x >= poi\_x and  wall\_min\_y <= poi\_y and  wall\_max\_y >= poi\_y:  dists.append(m)   dist = min(dists)  prob \*= probability(measurement, dist)  return prob  def probability(z, dist):  diff\_dist = z - dist  return math.exp((- (diff\_dist \*\* 2)) / (2\*(SIGMA \*\* 2))) + K |

(b)

Since we have a function above which takes 20 measurements, we can assume we use this.

Set 100 particles at the start position of the robot, each with weight 1/100.

Take a measurement from the sonar every 0.1 seconds.

Every 2 seconds, call the algorithm above with the previously taken 20 values on each particle in order to update the weight.

Normalise these weights so they add to 1.

Resample the particles based on these new weights.

**Per the revision lecture:**

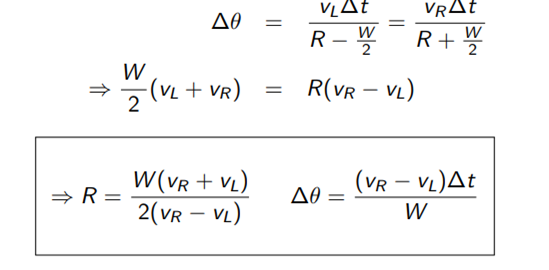
* Assume robot has sufficiently powerful CPU
* Repeatedly perform re-sampling as you go along, taking the current measurement and adding random deviation to your particles.
* E.g. We assume sensor points at some angle 20 degrees, and has moved X distance.
* Perform the random update for X distance
* Perform resampling based on measurement

c)

* Try and rotate the sensor to point at a right angle relative to you (based on estimated position)
* Try and read straight ahead, assuming you’re travelling towards a wall, continuously reducing measurements will squash your uncertainty in that direction.
* Based on estimated position, read the closer wall as it should be more accurate
* (Revision lecture) Prioritise scans which would reduce the spread of particles. I.e. Driving at a wall will give less spread in that direction, but spread will increase perpendicular to direction of travel. Scanning perpendicular to the direction of travel (or close to) will decrease the spread in that direction.x

Q3

(a)

iST\h

(b) §

R = W\*(V\_R + random.gauss(0, sigma\_v) + V\_L + random.gauss(0, sigma\_v))

/ 2\*(V\_R + random.gauss(0, sigma\_v) - V\_L + random.gauss(0, sigma\_v))

DeltaTheta = DeltaT\*(V\_R + random.gauss(0, sigma\_v) - V\_L + random.gauss(0, sigma\_v)) / W

Alternative solution?:

e = random.gauss(0, sigma\_v)

f = random.gauss(0, sigma\_v)

R = W\*(V\_R + e + V\_L + f) / 2\*(V\_R + e - V\_L - f)

DeltaTheta = DeltaT\*(V\_R + e - V\_L - f) / W

(c)

def predictState(vL, vR, delta\_t):

for particle in particleList:

vRWithError = vR + random.gauss(0, sigma\_vl)

vLWithError = vL + random.gauss(0, sigma\_vr)

RNumerator = W \* (vRWithError + vLWithError)

RDenominator = 2 \* (vRWithError - vLWithError)

R = RNumerator / RDenominator

delta\_theta = ((vRWithError - vLWithError) \* delta\_t) / W

particle.x += R(math.sin(delta\_theta + particle.theta) − math.sin(particle.theta))

particle.y -= R(math.cos(delta\_theta + particle.theta) − math.cos(particle.theta))

particle.theta += delta\_theta\*

Q4

(a)

average\_velocity = 10

distance = 30

k\_p = ?

while True:

z = read\_sonar\_corrected()

delta\_v = k\_p \* (distance - z)

# We v\_L > v\_R when distance - z > 0, and vice versa.

# I think we need to clip to stay within the bounds of the velocity functions.

set\_left\_velocity(clip(average\_velocity + delta\_v, 0, 20))

set\_right\_velocity(clip(average\_velocity - delta\_v, 0, 20))

time.sleep(0.05)

D

where

def clip(n, lo, hi):

return max(lo, min(n, hi))

#global m

m = []

def read\_sonar\_corrected():

# Take at least one reading

m.append(GetSonarDepth())

# Fill in if empty

While len(m) < 5:

m.append(GetSonarDepth())

time.sleep(0.01)

While len(m) > 5:

m.pop(0) # Remove old entries

return median(m)

\_\_\_\_\_

Alternative using a queue:

|  |
| --- |
| from numpy import median  avg\_v = 10 dist = 30 k\_p = ? readings = []  for i in range(5):  queue  while(True):  queue = queue[1:]  queue.append(GetSonarDepth())  z = median(queue)  delta\_v = k\_p \* (dist - z)  SetLeftVelocity(clip(avg\_v + delta\_v, 0, 20))  SetRightVelocity(clip(avg\_v - delta\_v, 0, 20))  time.sleep(0.05)  def clip(n, lo, hi):  Return max(lo, min(n, hi)) |
|  |

(b)



A high proportional gain results in a large change in the output for a given change in the error. If the proportional gain is too high, the system can become unstable (see the section on loop tuning). In contrast, a small gain results in a small output response to a large input error, and a less responsive or less sensitive controller. If the proportional gain is too low, the control action may be too small when responding to system disturbances.

The integral term accelerates the movement of the process towards setpoint and eliminates the residual steady-state error that occurs with a pure proportional controller. However, since the integral term responds to accumulated errors from the past, it can cause the present value to overshoot the setpoint value (see the section on loop tuning).

Derivative action predicts system behavior and thus improves se

ttling time and stability of the system.

(c)

Replace the line

delta\_v = k\_p \* (distance - z)

with

delta\_` = calculate(distance, 30)

integral = 0

previous\_error = 0

...

def calculate(setpoint, measured\_value):

error = setpoint - measured\_value

integral = integral + error \* d\_t

derivative = (error - previous\_error) / d\_t

output = k\_p \* error + k\_i \* integral + k\_d \* derivative

previous\_error = error

return output

(c) Alternative:

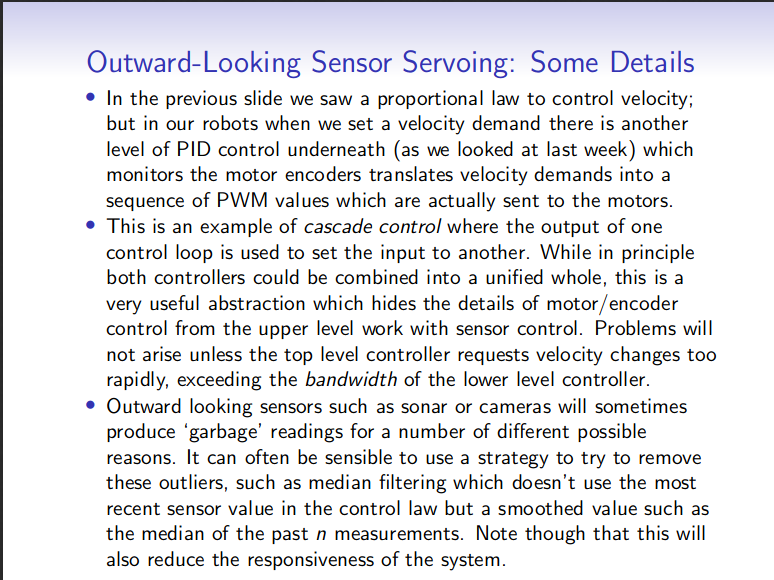
errors = [];  
def integrateError():

totalIntegral = 0  
 allErrors = errors.copy()  
 for time in range(1, len(allErrors)):  
 errorStep = allErrors[time]  
 priorStep = allErrors[time - 1]  
 totalIntegral += (errorStep \* 0.05) + ((errorStep - priorStep) \* 0.05 \* 0.5)  
 # ^ ASSUME as time step is so small, per step increment = rectangle + triangle  
 return totalIntegral

def calculate (setpoint, measured\_value):

error = setpoint - measured\_value;  
 errors.append(error)  
 update = k\_p \* error + k\_i \* integrateError() + k\_d \* (0 if errors.len() == 1 else ((error - errors[.len() - 2]) / 0. errors 05))

return update

(d)

Lecture 3 slide 20

So basically nested servvoing. Convenient because it allows the user to set the outer feedback loop value they’re concerned about and not worry about the inner feedback loops.