eCorrect these answers as necessary.

1a) Idea: In a while loop (while True) we check if d == 30. If so set left and right velocities at 10. Else if d < 30 we set left to 10 and right to 10 \* (30 - sonar) to adjust proportionally. Else set left to 10 \* (30 - sonar) and right to 10. Sleep for say ⅕ second to ensure we have less than 20 commands a second (to enforce max 20Hz).

|  |
| --- |
| 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) #(Andrew says its fine to not count execution time) |

where

|  |
| --- |
| def clip(n, lo, hi):  return max(lo, min(n, hi))  def read\_sonar\_corrected():  m = []  for i in range(5):  m.append(GetSonarDepth())  time.sleep(0.01)  return median(m) |

Alternative:

while True:

readings = []

for i in range(5):

readings.append(GetSonarDepth())

time.sleep(0.02)

z = median(readings)

d = 30

kp = 0.2 # z-d would have to be shockingly bad for the velocity to exceed 20cm/s

if z == d:

SetLeftVelocity(10.0)

SetRightVelocity(10.0)

elif z > d:

# gives symmetric behaviour so it looks smooth af

SetLeftVelocity(10.0 - 0.5 \* kp \* (z - d))

SetRightVelocity(10.0 + 0.5 \* kp \* (z - d))

elif d > z:

SetLeftVelocity(10.0 + 0.5 \* kp \* (d - z))

SetRightVelocity(10.0 - 0.5 \* kp \* (d - z))

time.sleep(0.2)

Note on the above solution:

If you define the error as z-d, you remove the need for three ifs.

[start of func]

v\_constant = 10

error = z-d

# assuming sensor is pointing 90 degrees to the left, if it’s 90 degrees to the right, swap # the input for SLV to SRV and vice versa

SLV(v\_constant - kp\*error)

SRV(v\_constant + kp\*error)

[rest of func]

b)

* e(t) is the difference between the desired and actual position. Increasing kp allows power to increase more quickly as e(t) increases.
* de(t)/dt is the rate of change of e(t) [velocity if e(t) is a distance or acceleration if e(t) represents a difference in velocity].   
  If de(t)/dt is negative, e.g. when robot is close to the destination and e(t) is decreasing, increasing kd reduces the speed of the robot quicker so it stops sooner. I.e. increase k\_d to reduce settling time
* The remaining integral term represents the history of all error terms. Increasing ki increases power based on the difference in error terms between initial error and the last known error to ensure that for small e(t) we have enough power to reach the destination. I.e. increase k\_i to reduce steady-state error

c) Idea: retain proportional control part but also track array of error terms and the time they were recorded at (two separate arrays). Setting ki and kd arbitrarily. We can calculate e(t) - e(t0) and multiply by some ki, maybe 2. e(t) is a distance so de(t)/dt will be a velocity. We could find the time taken to read the sonar reading (time.time() beforehand and afterwards) and take the difference and then speed = distance/time. Could set kd = 2. We also keep track of the current time by incrementing time by the time we sleep (assuming the other commands run instantly) and we add the current time to the array of time measurements at the end of each loop before we increment.

Attempt at code from idea above:

d= 30

cnt = 1

times = [time.time()]

err = [GetSonarReading() - d]

While True:

Z = GetSonarReading()

err.append(z-d)

K\_p\_term = k\_p \* err[cnt]

times.append[time.time()]

K\_i\_term += k\_i \* (err[cnt] - err[cnt-1])\*(times[cnt] - times[cnt-1])

K\_d\_term = k\_d \* (err[cnt] - err[cnt-1])/(times[cnt] - times[cnt-1])

u\_t = k\_p\_term + k\_i\_term +k\_d\_term

SetLeftVelocity(10 - u\_t)

SetRightVelocity(10 + u\_t)

cnt += 1

time.sleep(0.2)

d)

See lecture 4 slide 3.

One sensor is a problem if we are at a non 90deg angle, which can occur as we turn. Since sonar measurements become unreliable at non perpendicular angles. What's more, it does not give us the least distance from the wall.

Omnidirectional sensors are much better, as we can guarantee at least one sensor is perpendicular. And this sensor is guaranteed to be the closest distance.

Therefore the ‘simple strategy’ is taking the minimum reading. And use this for our error correction algorithm.

2a) Idea: using Wx, Wy, x and y, we determine the angle between them, alpha (math.atan2(Wy - y, Wx - x)) [arctan(Wy - y, Wx - x)] and the distance between them (sqrt((Wx - x)^2 + (Wy - y)^2))) and determine the angle to rotate (taking angles between - pi and pi):

If Wx - x > 0 then our new bearing is pi/2 - alpha. Otherwise if Wx - x < 0 our bearing is -pi/2 - alpha. Otherwise (Wx - x == 0) we consider Wy - y. If Wy - y > 0 then the new bearing is 0, else our new bearing is pi.

We then do new\_bearing - old\_bearing as our angle to rotate.

\_\_\_

|  |
| --- |
| def NavigateToWaypoint(Wx, Wy, CurrentState):   diff\_x = Wx - currentState.x  diff\_y = Wy - currentState.y  angle = math.atan2(diff\_y, diff\_x) - currentState.theta  dist = math.sqrt(diff\_x\*\*2 + diff\_y\*\*2)   if angle < -math.pi:  angle += 2 \* math.pi  if angle > math.pi:  angle -= 2 \* math.pi   Rotate(angle)  DriveForward(dist)   currentState.x = Wx  currentState.y = Wy  currentState.theta = angle   return currentState |

b) x\_pos = 0, y\_pos = 0, currentState = State(0, 0, 0)

while True:

# Ensures we always increment by w so the spiral increases outwards

x\_pos += w

currentState = NavigateToWaypoint(x\_pos, y\_pos, currentState)

y\_pos = x\_pos

currentState = NavigateToWaypoint(x\_pos, y\_pos, currentState)

x\_pos = - x\_pos

currentState = NavigateToWaypoint(x\_pos, y\_pos, currentState)

y\_pos = - y\_pos

currentState = NavigateToWaypoint(x\_pos, y\_pos, currentState)

x\_pos = -x\_pos

Alternative to 2b)

x, y, i = 0 # not sure if i even need to set x and y here if i just reset them in them in loop

cS = (0, 0, 0)

while True:

‘’”’

idea is make relative movements. By not returning and changing the currentState, this is achieved (currentState is always considered 0). Then the spiral is a lot easier to draw.

I.e. move x = +W, then y = +W, then x = -2W, then y = -2W, then x = +3W, then y = +3W, etc. so you end up sweeping out one extra W each turn. Used this sequence to calculate d below. I have written this solution assuming no walls/objects (would prob need to add this, i just haven’t yet)

‘’”’

d = ((-1) \*\* n) \* W \* (i + 1)

x = d

y = 0

NTW(x, y, cS)

x = 0

y = d

NTW(x, y, cS)

i += 1

Alternative to 2b)

state = State(0,0,0)

d = W

direction = 1

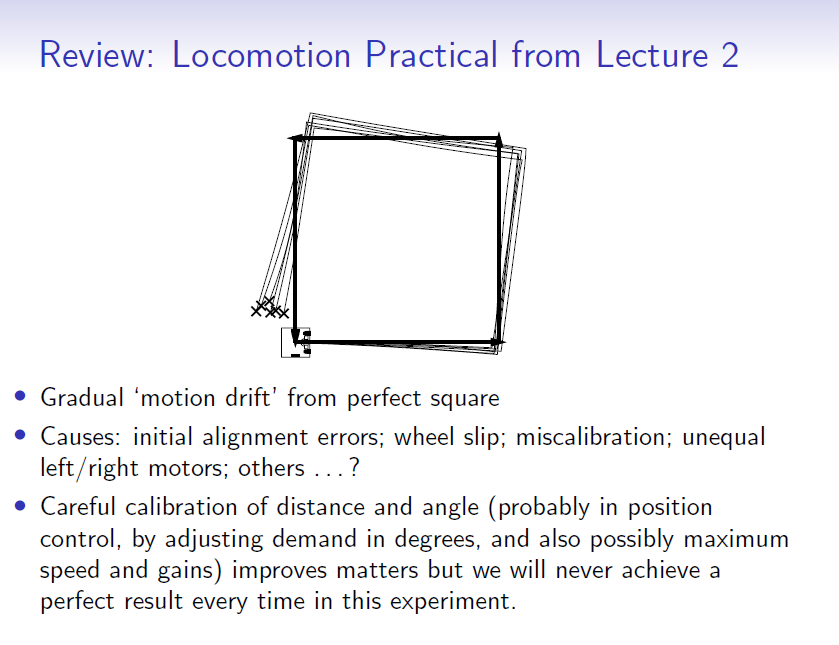
while True:

state = NTW(state.x + direction \* d, state.y, state)

state = NTW(state.x, state.y + direction \* d, state)

d += W

direction \*= -1

c) Something similar to the review of lecture 2, but for the spiral:

3a) Idea: Particle class with fields x, y, theta, weight. Have a particles array called particles, N = 100. InitialiseParticleSet creates 100 particles with (100, 100, 0, 1/N)

Andrew provides an example here: <https://www.doc.ic.ac.uk/~ajd/Robotics/RoboticsResources/particleDataStructures.py> but tbh it’s not that great.

particleList = [Particle] \* 100

N = 100

class Particle:

\_\_init\_\_(self, x, y, theta, weight):

self.x = x

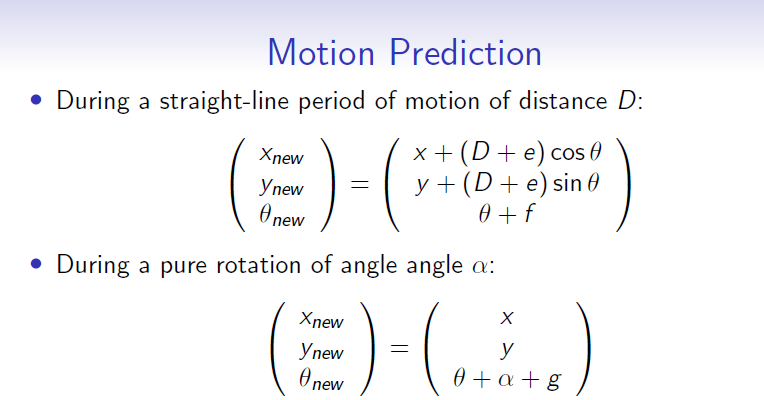
self.y = y

self.theta = theta

self.weight = weight

def func InitialiseParticleSet():

particleList = [Particle(100, 100, 0, 1/N) for i in range(N)]

b) Idea: if alpha = 0 then apply top formula, otherwise apply bottom formula. I took e as random.gauss(0, 2), f as random.gauss(0, 1) and g as random.gauss(0, 3) as I assume the angle of rotation will be smaller when attempting to move in a small line and am accounting for the uncertainty of movement over the current surface.

c) Idea: use math.exp(-(z-m)\*\* 2)/ (2 \* sigma ^ 2) + K to calculate likelihood per particle, and take K as some small number (say 0.01). We then multiply the weight of each particle by the likelihood.

The m here is the closest ceiling distance for each particle

d) Idea: sum weights together and divide all particles’ weights by the sum calculated.

e) Idea: use CDFs. We go over all particles and find their weight. We track a variable, curr\_weight, which we add the weight of each particle to as we pass it. We then append the new value of curr\_weight to an array, cdfs, as we check each particle. We track a list of new particles, new\_particles.

Then we take 100 random numbers between 0 and 1 using random.random() and for each one we go over the array cdfs and check whether the number is less than the current cdf. If it is we have found the correct bucket and we add particles[i] to new\_particles, i is current index at cdfs. Then at the end we set particles = new\_particles.

Copied answer below from 2017/18 paper:

def ResampleParticleSet():

# Generate cumulative weights, assuming len(particleList) > 1.

cumulative\_weights = []

sum = 0

for particle in particleList:

sum += particle.weight

cumulative\_weights.append(sum)

new\_particles = []

for \_ in range(len(particleList)):

# Threshold for intersection per slides.

cumulative\_threshold = random.random()

# Could use bisect here instead to optimise in real world case.

i = 0 # Index of particle to be selected.

while (cumulative\_threshold >= cumulative\_weights[i]):

i++

# Set the new particle at this index.

selected\_particle = particleList[i]

selected\_particle.weight = 1 / len(particleList)

new\_particles.append(selected\_particle)

return new\_particles

4ai) It’s the probability of obtaining measurement “z” given that the ground truth value you have is “v”. Essentially, it’s a probability to reflect how reasonable our measurement is. “A likelihood function fully describes a sensor’s performance.”

ii) (I’m not sure what this question actually means) Take 5 readings from the sonar and choose then take the median of these 5 readings, so as to toss out garbage values. We do need to take into the account where a particle is so off that the sensor readings could all be bad (maybe due to the sensor being too far away from the wall). Then we just ignore this round of readings and move on. Repeat for a range of depths.

Also for different reflection angles?

iii) (idk what zero-mean errors mean)

Systematic errors would show as something like z = 1.1v for every value and so will always be off no matter what but it is something you can easily account for in calculations

The shape of the distribution essentially says that when the particles are closer to (0, 0, 0) then we are more likely to see that z and v are equal (or that they are more likely to be closer to each other). As z and v increase then the uncertainty becomes greater and so the s.d of the gaussian distribution increases.

The sensor’s accuracy is generally fairly consistently fine when the real-world values are between 10-150 (somewhat arbitrary cumulative\_current values as an example). Anything lower than 10 tends to just read as 10 and anything above 150 will read as 255. There may be a slight offset in the readings as well that should be taken into account for.

iv) ~~I guess the p(z|v) height would remain the same throughout but a height of zero for the first 5cm?~~

Height and width must be constant for all gaussians.

But, must have a means that are shifted across to sit on z = v + 5.

b) (could be completely wrong here)

Robust Likelihood for Sonar Update 
• A robust likehood function models the fact that real sensors 
sometimes report 'garbage' values which are not close to ground 
truth. Robust functions have 'heavy tails'. This can be achieved 
most simply by adding a constant to the likelihood function. The 
meaning of this is that there is some constant probability that the 
sensor will return a garbage value, uniformly distributed across the 
range of the sensor. 
p(zlm) 
z=m 
z 
p(zlm) oc e 
• The effect of a robust likelihood function in MCL is that the filter is 
less aggressive in 'killing off' particles which are far from agreeing 
with measurements. An occasional garbage measurement will not 
lead to the sudden death of all of the Darticles in good Dositions. 

I believe it would look something like this where the flat bits start at 10 and 200cm where K = 0.1

Alternative:

(same) but where K = 0.1/190 ~=0.0005

This is because the question says 10% = 0.1 of measurements return garbage values. So we should have a **total** garbage probability of 0.1 (the area), but the probability of a garbage value at any given point should be 0.0005 (i.e. height of garbage prob. baseline).

So sensor range \* garbage value rate = total garbage value likelihood

c) We wish to get P(Oi | z) = P(z | Oi) P(Oi) / P(z). This is the probability of occupancy of cell i after / given measurement z. Practically, P(z) is very hard to get but we can apply Bayes theorem in the same manner for P(Ei).

As such, P(Oi | z) = P(z | Oi) / P(z | Ei) \* P(Oi) / P(Ei) \* P(Ei | z).

We note that we merely just use the value of P(Oi | z) to compare to 0.5 to judge if a cell should be updated to be occupied or not. As such, we can simplify computation using the log odds form (done below in other answer), as long as we also update our comparison value 0.5 as well to compare to.

Previous answer: something to do with lecture 7 but I don’t understand it enough

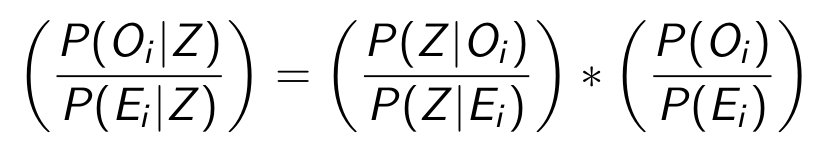
We know by bayes theorem that P(Oi | z) = P(z | Oi) P(Oi) / P(z)

We also know the same for P(Ei).

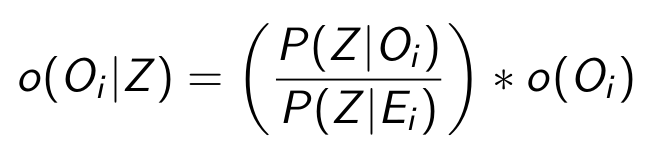
We know that P(Oi) + P(Ei) = 1 and thus Ei is the complement of Oi.

Odds form is o(A) = P(A) / P(!A)

Thus by dividing the posterior probabilities for Oi and Ei we get:



Which we can express the LHS and far RHS (not the middle) in odds form as:



And finally, we take logs to get the additive property

