**Hi Guys, I am the previous year student. It is nice to hear that this doc can still help. But only a few people in my year were involved in posting the past paper ans here, so in the end I did not pay much attention either. Those were all my “FIRST”-Version answers and no guarantee its correctness. Good Luck. Andy is a pretty nice lecturer and the exam is not hard at all, similar to past papers and what you had done during the tutorials. All I should say is, taking care about the details. (Every small mistake will result in a deduction....)**

t= Question 1:

(a)

|  |
| --- |
| *N = 100 # number of particles def CalculateMeanEstimate(particleList):  x\_sum, y\_sum, theta\_sum = 0.0, 0.0, 0.0  for item in particleList:  x\_sum += item.x \* item.w  y\_sum += item.y \* item.w*  *# Need to wrap-around angles or use atan2 mean  # 1: Wrap-around (this might still fail, eg with 0 and 2pi)  theta\_sum += (2 \* math.pi + item.theta) \* item.w  # Then normalise to -pi/pi in the end  theta\_sum = math.atan2(math.sin(theta\_sum), math.cos(theta\_sum))*  *# 2: Use angular sum  sin\_sum += math.sin(item.theta) \* item.w  cos\_sum += math.cos(item.theta) \* item.w*  *# Then normalise after loop  theta\_sum = math.atan2(sin\_sum, cos\_sum)   return x\_sum, y\_sum, theta\_sum* |
|  |

*192*

(b)

|  |
| --- |
| def DriveToWaypoint(Wx, Wy, xMean, yMean, thetaMean):  xto = Wx - xMean  yto = Wy - yMean  D = min( 0.2, math.hypot(xto, yto) ) # vague question wording, if min is needed  alpha = math.atan2(yto, xto) - thetaMean  alpha -= 2 \* math.pi if alpha > math.pi else 0  alpha += 2 \* math.pi if alpha < - math.pi else 0  RotateRobot(alpha)  DriveRobotForward(D)  return D, alpha |

(c)

# three uncertainties (could be calibrated later)

# as units in meters and radians

# the standard deviation of uncertainties is initialized to be 0.01m, 2\*pi/360, 2\*pi/360, respectively.

std\_e, std\_f, std\_g = 0.01, 2\*pi/360, 2\*pi/360

**# Per lecture 5: Variance of the gaussian distribution should scale with rotation angle and travelled distance**

def *MotionPrediction(D, alpha):*

*for item in particleList:*

*e =* ***random.gauss(0, std\_e \* math.sqrt(D))***

*f =* ***random.gauss(0, std\_f \* math.sqrt(D))***

*g =* ***random.gauss(0, std\_g \* math.sqrt(alpha))***

*# pure rotation*

*If (alpha > 0):*

*item.theta += alpha + g*

*# pure translation*

*If (D > 0):*

*item.x += (D + e) \* cos(item.theta)*

*item.y += (D + e) \* sin(item.theta)*

*item.theta += f*

***Alternative***

def MotionPrediction(D, alpha):

for p in particleList:

e = random.gauss(0, std\_e + math.sqrt(D))

f = random.gauss(0, std\_f + math.sqrt(D))

g = random.gauss(0, std\_g + math.sqrt(alpha))

p.theta += alpha + g

p.x += (D + e) \* math.cos(p.theta)

p.y += (D + e) \* math.sin(p.theta)

p.theta += f

(d)

|  |
| --- |
| sigma\_s = 2 # uncertainties of the sensor (calibrate it in experiment) def MeasurementUpdate(z):  k = 0.01  for item in particleList:  dist = GetDistanceToWall(item.x, item.y, item.theta)  item.weight \*= math.exp(- (z - dist)\*\*2 / (2 \* sigma\_s \*\*2)) + k |

(e)

|  |
| --- |
| def NormaliseParticleSet( ):  sum\_of\_weights = 0.0  for item in particleList:  sum\_of\_weights += item.weight  for item in particleList:  item.weight /= sum\_of\_weights |

(f)

|  |
| --- |
| 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(particle(selected\_particle.x, selected\_particle.y, selected\_particle.theta, 1/100))   particleList = new\_particles |
|  |

Question 2:

(a)

|  |
| --- |
| def calculateObstacleDistance (x, y, theta):  min\_d = float(“inf”) # or maybe a sufficiently large number, like 100  for item in obstacles:  d = math.sqrt((item.x - x)\*\*2 + (item.y-y)\*\*2) - 0.1 - 0.15  if d < min\_d:  min\_d = d  If min\_d < 0:  return -1  else:  return min\_d |

(b)

(c)

Purely local planning may drive the robot to a deadlock and get stuck.

While global planning such as wavefront method or rapidly exploring randomized trees method can resolve this problem by breadth first search the entire environment or growing a tree of connected nodes by random sampling, respectively.

Question 3:

(a)

# <- Explains Vr, Vl

|  |
| --- |
| ***sL = list() sR = list() def stay\_mid():  time\_start = now()  if len(sL)>=5:  sL.pop(0) # Remove old (first) value  sR.pop(0)  sL.append(getSonarMeasurements()[0])   sR.append(getSonarMeasurements()[1])   L = median(sL)  R = median(sR)  delta\_V = K \* (L - R) # K is the tunable parameter  setWheelVelocities(0.5 - delta\_V, 0.5 + delta\_V)  elapsed\_time = now() - time\_start  if elapsed\_time < 0.1:  time.sleep(0.1 - elapsed\_time)*** |

time.sleep(0.1) <- Explains timing calculation

(b)

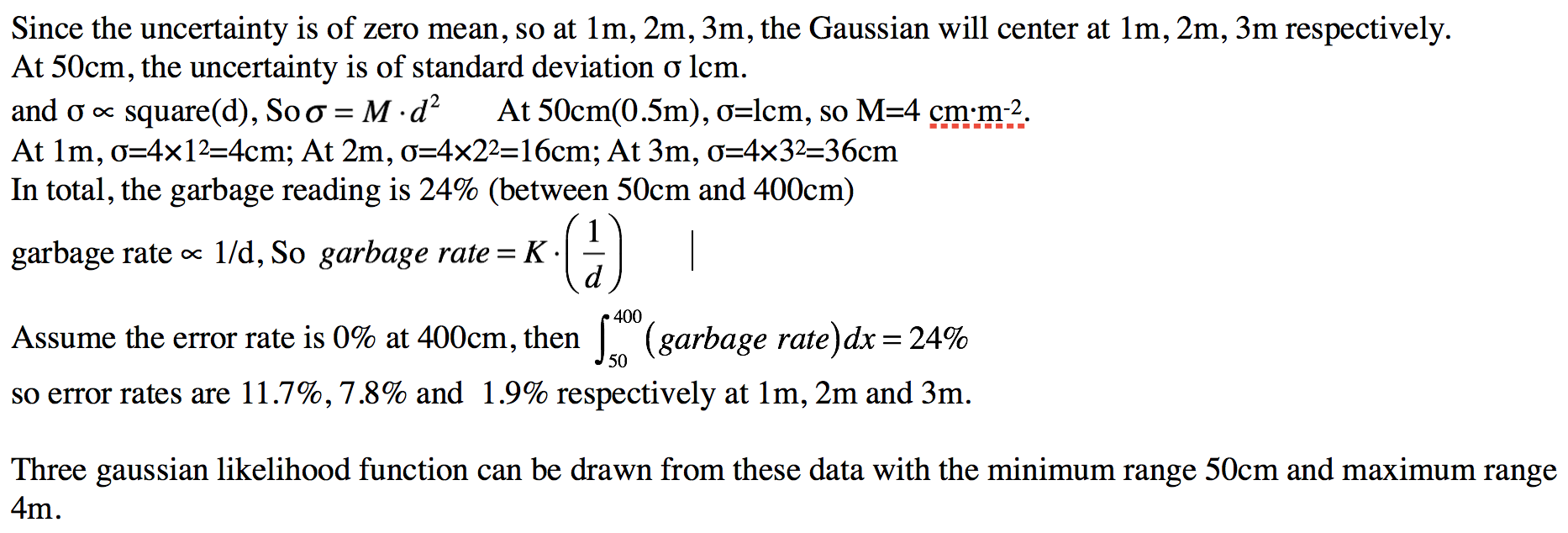
(This isn’t really using signatures but it seems the most obvious way)

When the robot is places, it can drive forward 1 metre and take readings for width. Since we know that the robot is only able to go in one direction, it essentially has one degree of freedom (like a train). This means we can use the measurements and scan the saved data to find a 1m section which matches out reading. We can use the sum of squared differences to test this. Since the tunnel is 50m, and we are going at 0.5m/s reading at a rate of 10Hz, we take a reading every 0.05m, and so will have 1000 readings. This means we need to do ~900 sum of squared differences calculations (which could be expensive, which is why we may use signatures at set intervals and just compared to these, it just would not be as accurate, but then we could do this approach around the closest matching signature, which would be less expensive).

Alternative (Using signatures)

During the initial learning phase,  
For the entire cave tunnel:  
 1. Follow the tunnel for 1m  
 2. Then start taking \*very\* small steps. Memorize the next location where the distance between the left or right wall has reached a local maximum or minimum. (You may have to walk back a step after realizing the prior position was a local maximum/ minimum)  
This serves as a trait the robot can identify on later.

3. repeat  
  
During the find phase, try:  
 1. Take \*very\* small steps.  
 Find a location where the distance between the left and right wall has reached a local maximum or minimum. (You may have to walk back a step after realizing the prior position was a local maximum/ minimum)  
Compare it with your signatures (Set a decently high threshold for rejection)  
 2. If it was a signature, you have found your location. Success  
 3. Else keep making small steps and repeating.

Question 4:

(a)

4a alternative [unfinished]:

So growing uncertainty as depth increases (because of sigma <proportional> d\*\*2) means our likelihood plots will decrease in height (and get wider?).

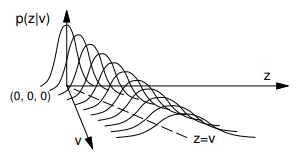
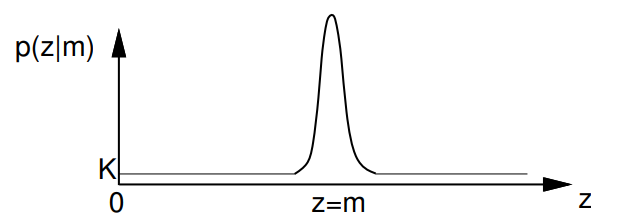
Garbage values decreasing as depth increases (because of garbage rate <proportional> 1/d) means that there will be a larger proportion of values that fit the Gaussian likelihood.

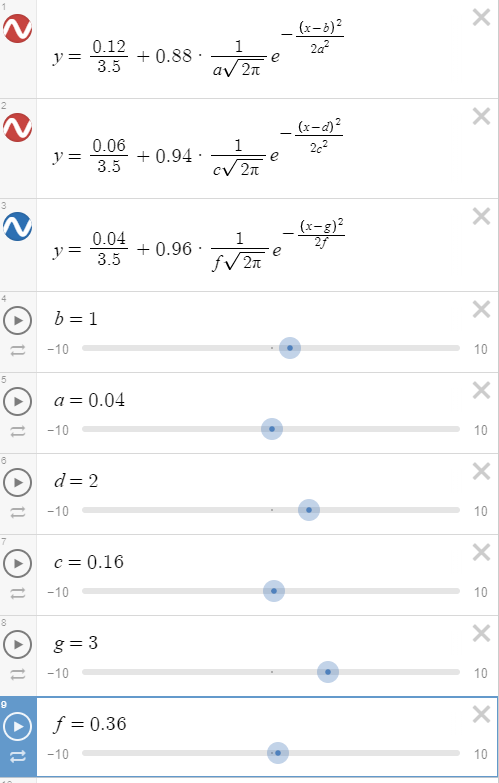
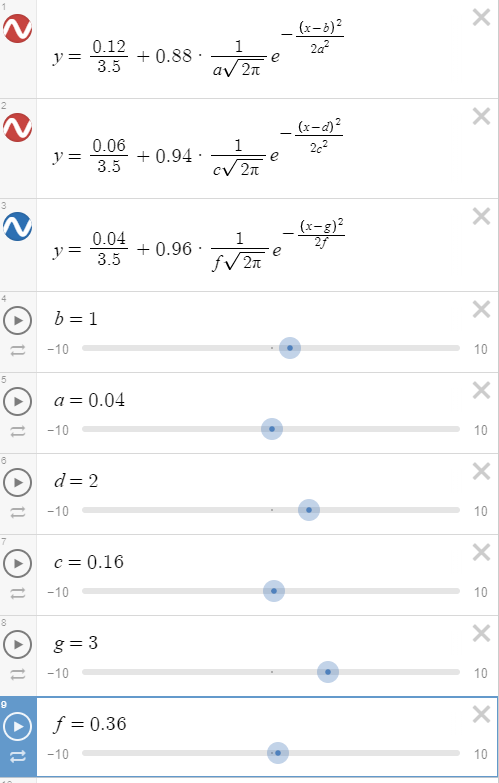
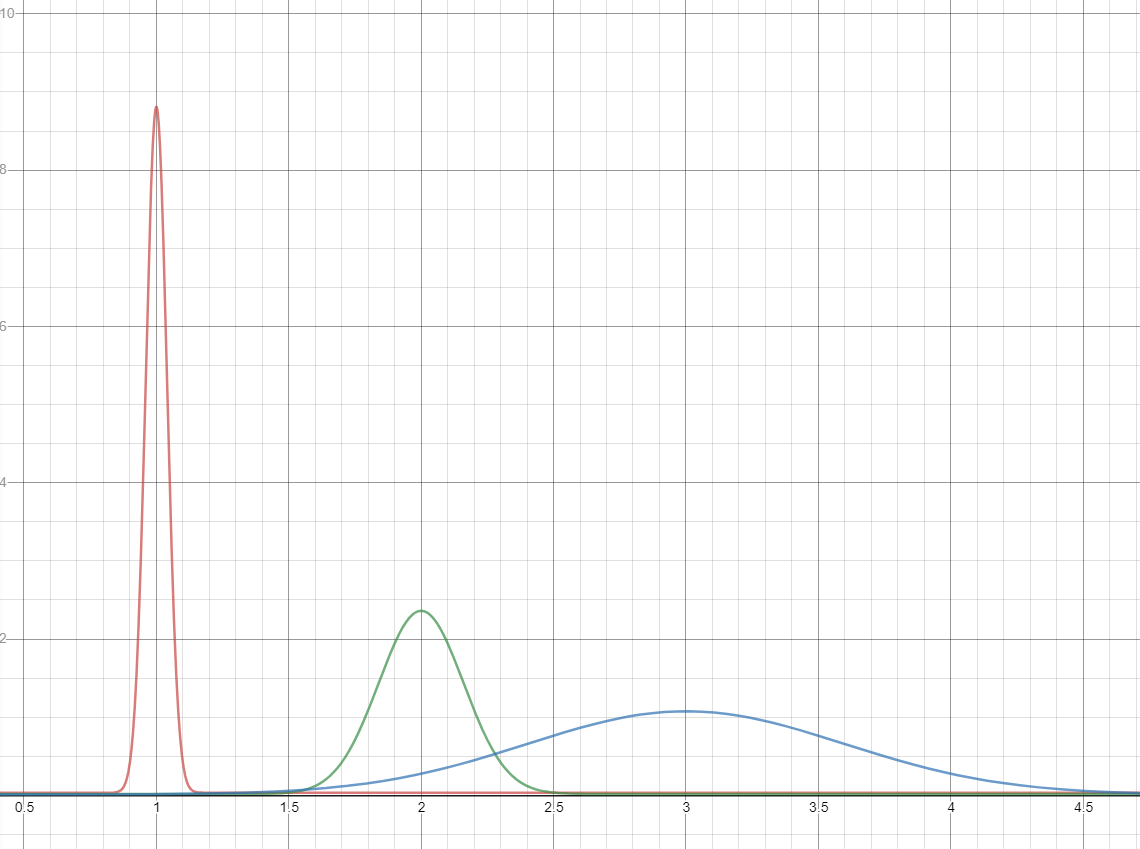
We then work out specific values to annotate our diagrams [i still don’t know how exactly we do annotate the uncertainty and garbage rate].

Plug in values we have (g = 0.24, d = 0.5m) to work out k. Solving the equation, we get k = 0.12.

We then use k to determine the garbage values of the other distances:

We do something similar with to get

I imagine something like this graph from the slides:  
  
Where k = g/3.5 (the whole range 50cm - 4m)

M is 1/2/3 metres (shift the graph from left to right for the slices)  
Do we have to give the density peak?  
Plotting it into desmos:

(b)

(i)

Arrows positioned all over the map, with arrows randomly oriented.

(ii)

Arrows facing a wall 30cm in front of it.

(iii)

Same as above but also arrows that have a wall 20cm to the right of their direction

(iv)

A magnetic compass sensor reports the bearing angle relative to north. The likelihood only depends on the orientation of the robot. Suppose the compass sensor is accurate, we can model the likelihood, depending on the difference between the measured(actual) bearing of robot relative to the north β, and the estimated one (calculated by the difference between bearing of the x-coordinate of world reference frame γ, which is known at the beginning, and robot current orientation θ) using Gaussian distribution. And then perform an update of particle weights, similar to the update based on sonar sensor. The particles far from the right orientation will get low weights or be killed.

Magnetic compass sensor can be used as an alternative of a ring of sonars.

Especially useful in rooms where many locations have similar sonar depth signatures (see 2018-19 Q1d)

$