Orthogonality

= 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*

*y\_sum += item.y*

*theta\_sum += item.theta*

*return x\_sum/100, y\_sum/100, theta\_sum/100*

(b)

def *DriveToWaypoint(Wx, Wy, xMean, yMean, thetaMean):*

D = math.sqrt((Wx-xMean)\*\*2 + (Wy-yMean)\*\*2)

*# calculate the arctan of* δy/δx with consideration of their signs

*alpha = math.atan2(Wy-yMean, Wx-xMean)*

*RotateRobot(alpha)*

*DriveRobotForward(D)*

(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

def *MotionPrediction(D, alpha):*

*for item in particleList:*

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

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

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

*# pure rotation*

*item.theta += alpha + g*

*# pure translation*

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

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

*item.theta += f*

(d)

sigma\_s = 2 # uncertainties of the sensor (calibrate it in experiment)

def MeasurementUpdate(z):

for item in particleList:

dist = GetDistanceToWall(item.x, item.y, item.theta)

item.weight = math.exp(- (z - dist)\*\*2 / (2 \* sigma\_s \*\*2))

(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)

N = 100 # number of particles

def ResampleParticleSet( ):

key = list()

lower\_index = 0.0

for item in particleList:

upper\_index = lower\_index + item.weight

key.append([(lower\_index, upper\_index), item])

new\_list = list()

# sample new particles

for i in range(0, 101):

p = random.random()

for item in key:

if p <= item[0][1] and p > item[0][0]: new\_list.append(item[1])

if p == 0.0: new.append(key[0][1])

# reweight

for item in new\_list:

item.weight = 1.0/N

f)

def resample(N):

for i in range (N):

w = random.random()

j = 0

while (w > particleslist[j].cumulative\_weight):

j += 1

newparticle = particlelist[j]

Question 2:

(a)

def calculateObstacleDistance (x, y, theta):

min\_d = 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

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)

def stay\_mid():

sL = list()

sR = list()

if len(sL)>=5:

sL.pop()

sR.pop()

sL.append(***getSonarMeasurements()[0]***)

sR.append(***getSonarMeasurements()[1]***)

L = median(sL)

R = median(sR)

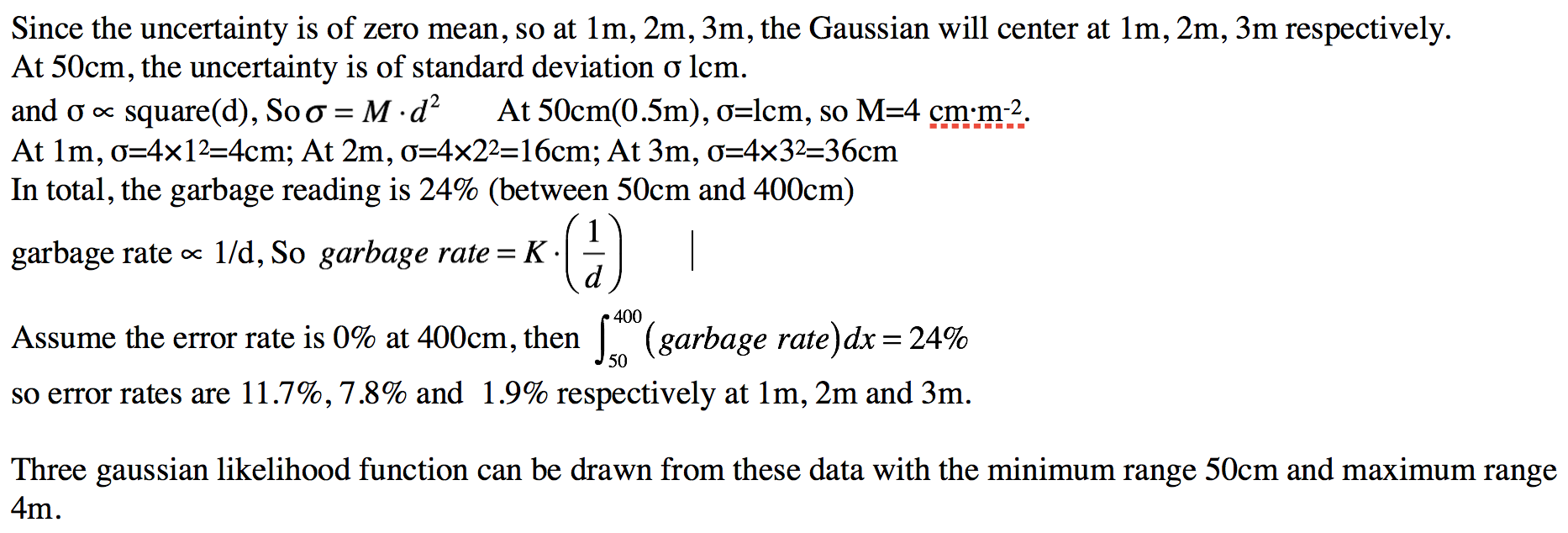
delta\_V = K \* math.abs(sL - sR) # K is the tunable parameter

***setWheelVelocities (vL + delta\_V, vR - delta\_V)***

(b)

Question 4:

(a)



(d)

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.