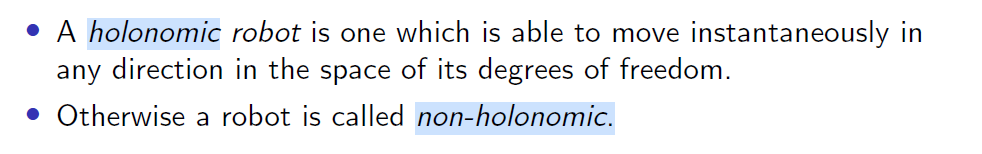
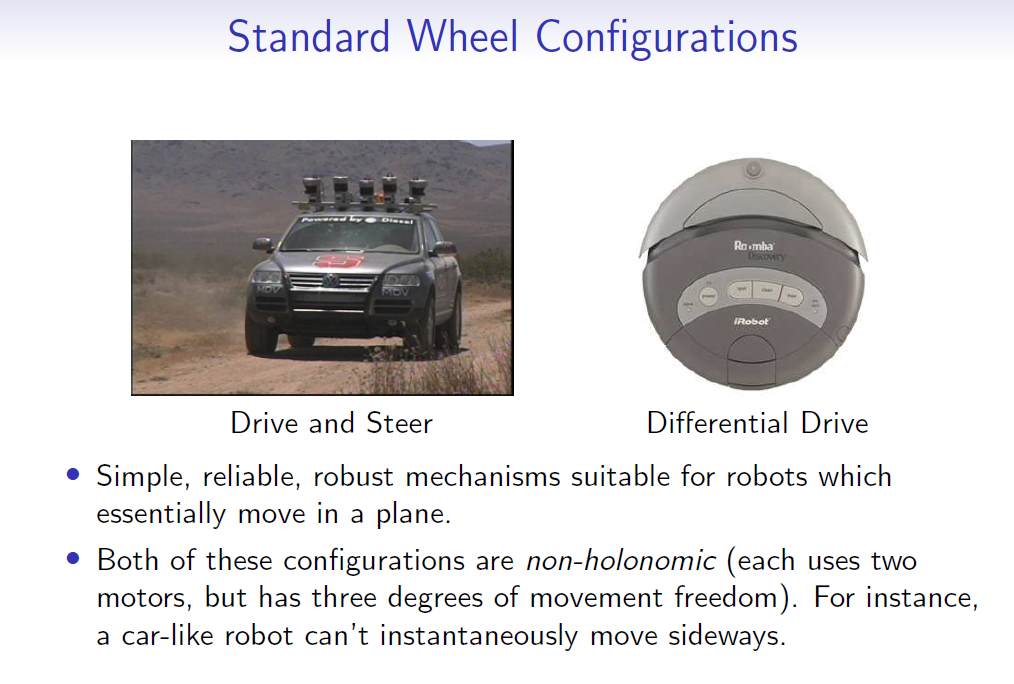
also idk any other way to do it lolCorrect these answers as necessary.

1a) Def holonomic from slides:





Now we consider all rotations to be clockwise/anticlockwise with respect to a point somewhere in the top left of the page.

Since considering motions to be wrt to the robot, disagrees with the equations laid out - If the front 2 wheels were both clockwise **wrt to robot**, it would actually spin. So we must use a global point instead.

Note also **positive** angular velocities correspond to **anticlockwise** rotation.

To move straight forward, set w1 = w2 = w3 = w4 = 1.

To slide left, use w1 = w4 = -1, w2 = w3 = 1.

To rotate on spot use w1 = w3 = -1 and w2= w4 = 1.

1b) Idea: We move forwards, scaling either w1 and w2 or w3 and w4 as we move. This would allow us to turn in an arc without rotating on the spot. We can use wr = u / r (where u is the specified velocity) and then vRx = 1, vRy = 0.

We then check the angle alpha using GetAlpha().

Useful link

By multiplying by the angle converted to degrees, we ensure that we multiply by a large enough number (to avoid cases for small angles in radians either not turning or turning incorrectly).

To turn, set w2 and w4 by t \* alpha \* (180 / pi) (and set w1, w3 to the opposite value?)

Then we can multiply t by alpha \* (180/pi).

We then set all wis using SetOmega.

Place this in a while loop.

1b)

u = 1 # required velocity

kprop = 0.1 # proportional control constant

while True:

alpha = GetAlpha()

# left side wheels

# - must slow down if +ve alpha => object to the left

# - must speed up if -ve alpha => object to the right

omega\_1 = (1/r)(u - kprop \* alpha)

omega\_3 = (1/r)(u - kprop \* alpha)

SetOmega(1, clip(omega\_1, 0, 2 \* u))

SetOmega(3, clip(omega\_3, 0, 2 \* u))

# right side wheels

# - must speed up if +ve alpha => object to the left

# - must slow down if -ve alpha => object to the right

omega\_2 = (1/r)(u + kprop \* alpha)

omega\_4 = (1/r)(u + kprop \* alpha)

SetOmega(2, clip(omega\_2, 0, 2 \* u))

SetOmega(4, clip(omega\_4, 0, 2 \* u))

def clip(val, low, high):

return min(max(low, val), high)

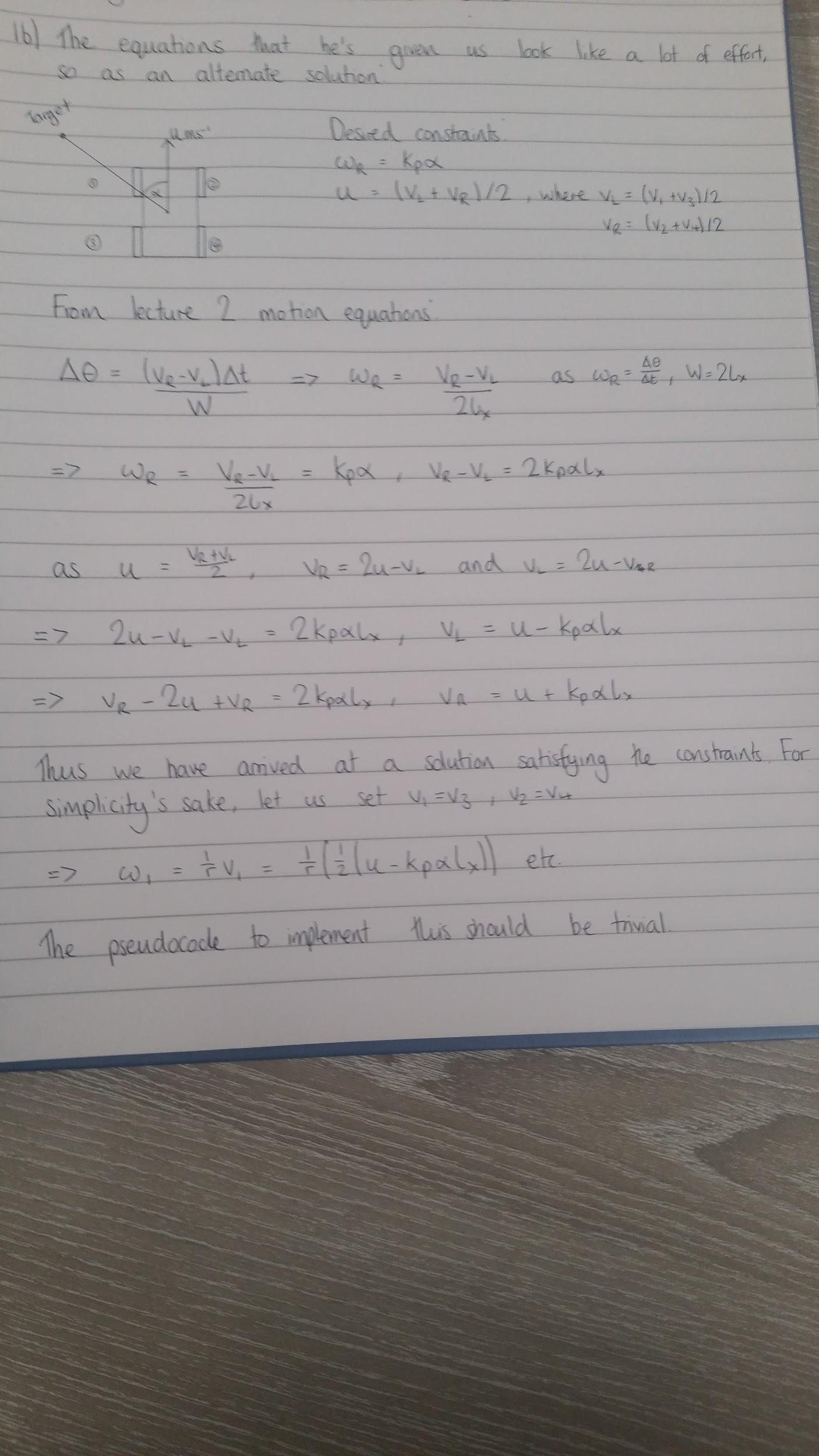
1b) Another alternative:

The question mentions **without sliding sideways,** so:

- We take V\_Ry to be 0.  
- V\_Rx needs to be some constant U

- W\_R is some angular velocity times the angle we need to be at.

|  |
| --- |
| U = Forward velocity K = angular velocity constant while True:  alpha = getAlpha()  w1 = 1/r \* (U - (lx + ly) \* K \* alpha)  w3 = 1/r \* (U - (lx + ly) \* K \* alpha)  w2 = 1/r \* (U + (lx + ly) \* K \* alpha)  w4 = 1/r \* (U + (lx + ly) \* K \* alpha)  SetOmega(1, w1)  SetOmega(2, w2)  SetOmega(3, w3)  SetOmega(4, w4) |



1c) Idea: use getD() and getAlpha() to find distance and angle. We need to maintain D - R dist. We can set kp = D - R and move forward by (D - R) \* cos alpha (alpha = getAlpha()) so we are proportionally increasing or decreasing speed. We can then slide by (D - R) \* sin alpha. Place this in a while loop so it continues to check until the robot has held position.

Alternative (with diagram & code)?:

Move forward by: (D \* cos(α)) - R

Slide left by: D \* sin(α)



R = 10 # required dist

fprop = 1

sprop = 1

+

while True:

alpha = GetAlpha()

D = GetD()

forwardness = D \* cos(alpha) - R

sidewaysness = D \* sin(alpha)

# proportional constants

forwardness \*= fprop

sidewaysness \*= sprop

# Signs of these according to q1a

SetOmega(1, 1/r(forwardness - sidewaysness))

SetOmega(2, 1/r(forwardness + sidewaysness))

SetOmega(3, 1/r(forwardness + sidewaysness))

SetOmega(4, 1/r(forwardness - sidewaysness))

2ai) The particles would sort of form a line where the width is increasing but particles that have too much variance in depth are cut off.This is due to particles being able to stray further away from the actual position as long as the depth from the wall is the same. Rotation shouldn’t have much effect.

ii) basically the same as the first part but in the other direction. Particles will increase along the direction in which the robot travels but not in the direction in which the sonar faces

iii) The particles would spread out and then every time there is a reading they will group together closer to the actual position of the robot in all directions.

2b) Idea: We calculate m using the formula in slides. (Could calculate numerator and denominator separately and if denominator = 0 then we skip over this wall). We find the x intercept and y intercept from slides and check if these coordinates would intercept the walls. If so then we check if the new distance m is less than a var we track, min\_dist (set to sys.maxsize or infinity initially) and if so update min\_dist with m.

We calculate the likelihood for each sonar measurement comparing z[i] to m using the formula from lectures and multiply these together to form a variable joint\_likelihood. We then return joint\_likelihood.

We also will need to take about 5 readings for each sonar and then take the median of those 5 readings to help eliminate “garbage” values

EPSILON = 0.2

STD = 2

ANGLE\_BETWEEN\_SONARS = 2\*math.pi / 12

SONAR\_OFFSET = R

"""

Calculates the likelihood for a single particle.

"""

def get\_likelihood(particle: Particle, z: List[float], walls: List[Wall]) -> float:

likelihood = 1

x, y, theta = particle

# Multiply likelihoods together, to get combined likelihood (from lecture).

for i, z\_i in enumerate(z):

m = calculate\_distance\_to\_wall(x, y, theta + i\*ANGLE\_BETWEEN\_SONARS, walls)

likelihood \*= likelihood\_function(z\_i + SONAR\_OFFSET, m)

return likelihood

"""

Calculates the distance of a particle to the wall that it's facing.

"""

def calculate\_distance\_to\_wall(x:float, y: float, theta: float, walls: List[Wall]) -> float:

min\_distance = float('inf')

for wall in walls:

a\_x, a\_y, b\_x, b\_y = wall

numerator = (b\_y - a\_y) \* (a\_x - x) - (b\_x - a\_x) \* (a\_y - y)

denominator = (b\_y - a\_y) \* math.cos(theta) - (b\_x - a\_x) \* math.sin(theta)

# Should probably use an epsilon check here.

if denominator == 0:

continue:

distance\_to\_wall = numerator / denominator

if distance\_to\_wall < 0:

continue

intersection\_x = x + distance\_to\_wall \* math.cos(theta)

intersection\_y = y + distance\_to\_wall \* math.sin(theta)

if not point\_lies\_on\_wall(intersection, wall):

continue

# Maybe add some logic to deal with low angle-of-incidence readings.

min\_distance = min(min\_distance, distance\_to\_wall)

return min\_distance

"""

Checks that point with coordinates (p\_x, p\_y) is on wall defined by (a\_x, a\_y) and (b\_x, b\_y).

"""

def point\_lies\_on\_wall(p\_x, p\_y, a\_x, a\_y, b\_x, b\_y) -> bool:

distance\_a\_b = distance((a\_x, a\_y), (b\_x, b\_y))

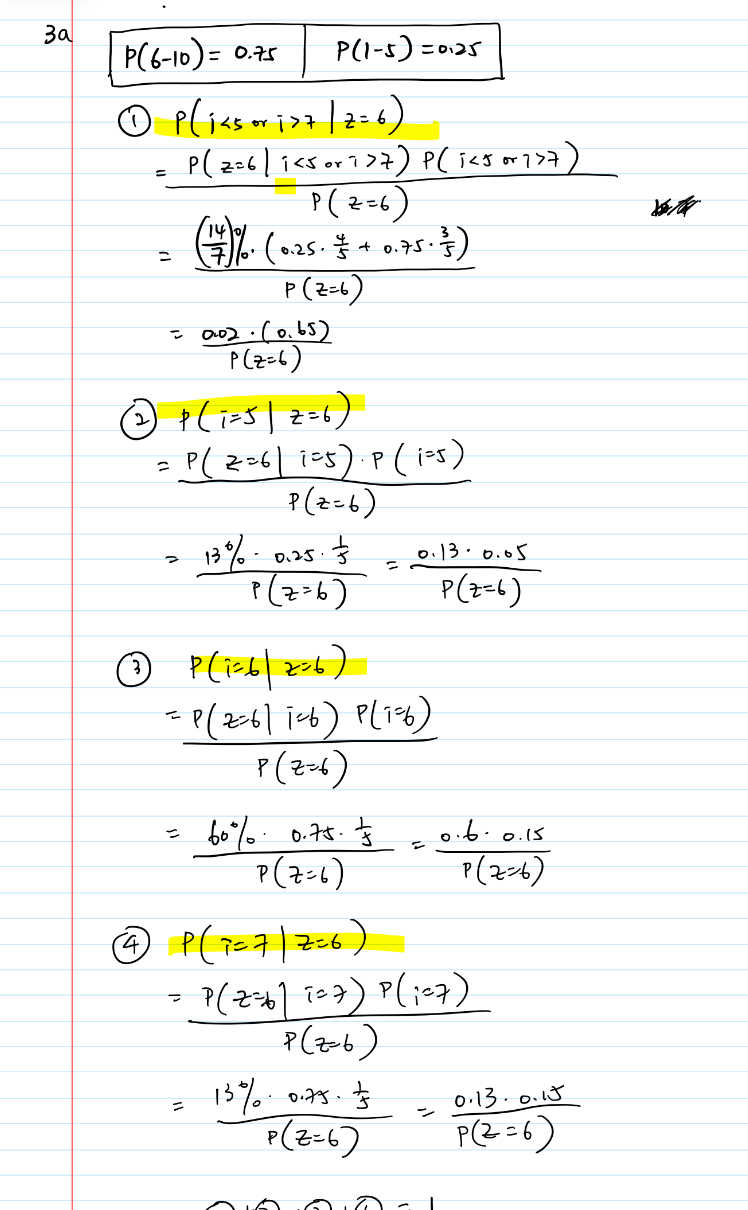
distance\_a\_p = distance((a\_x, a\_y), (p\_x, p\_y))

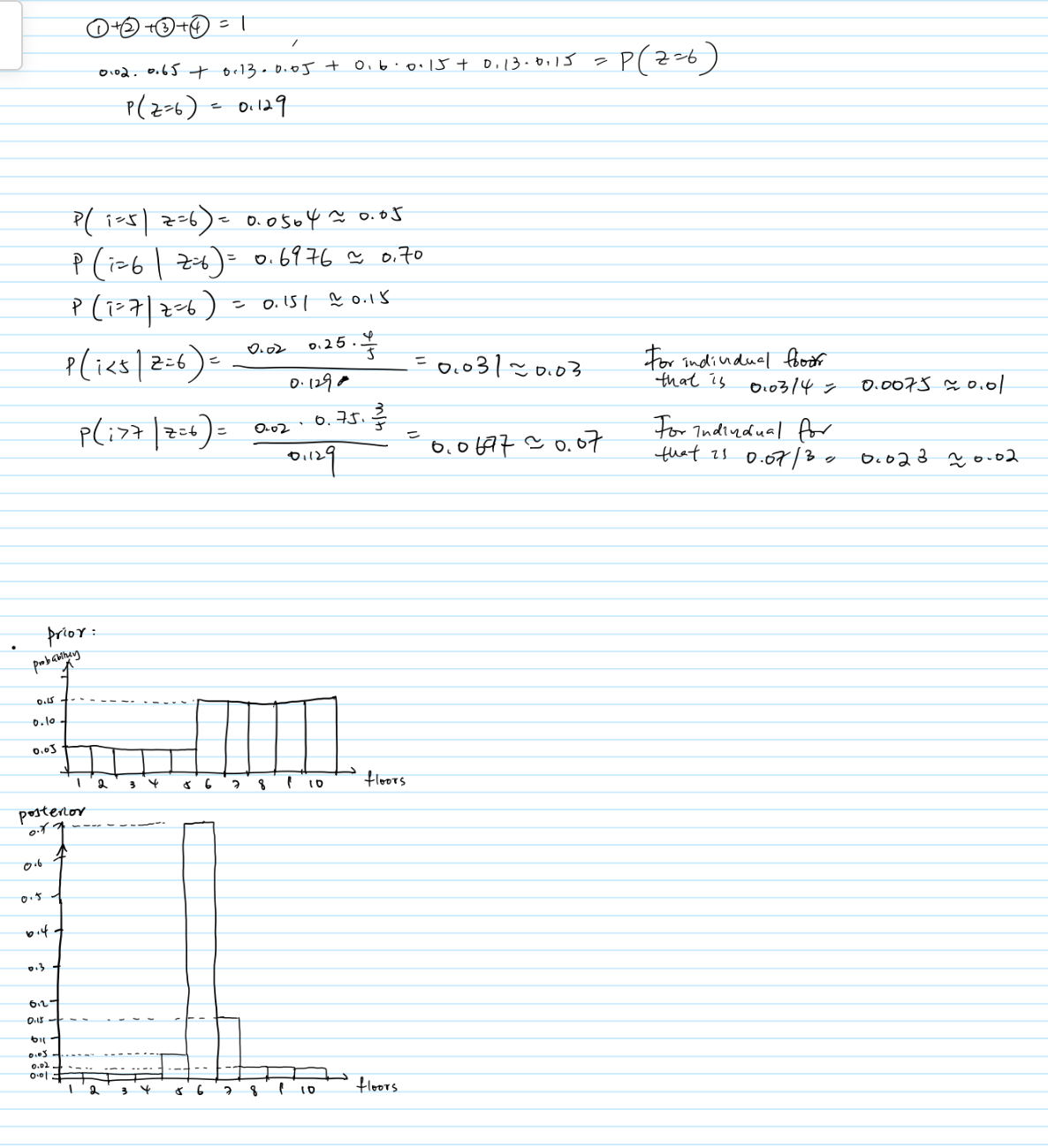
distance\_p\_b = distance((p\_x, p\_y), (b\_x, b\_y))

return abs(distance\_a\_p + distance\_p\_b - distance\_a\_b) < EPSILON

def likelihood\_function(z: float, m: float) -> float:

return math.exp(-(z-m)\*\*2/(2\*STD\*\*2))



W

b) There is some systematic error in the 20, 40 and 60 readings of about 10cm (maybe some calculations would give a more accurate error but idk if that’s important)

Get the mean of the readings and calculate the standard deviation from this

Then use

Likelihood for Sonar Update 
• The likelihood should depend on the difference z — m: if this is 
small, then the measurement validates a particle; if it is big it does 
not and weakens the particle. 
• The further away the measurement is from the prediction, the less 
likely it is to occur. A Gaussian function is usually a good model for 
the mathematical form of this. The standard deviation as is based 
on our model of how uncertain the sensor is, and may depend on z 
or may be constant. 
p(zlm) oc e 
20 
• Note the difference between this and the motion prediction step. 
There we sampled randomly from a Gaussian distribution to move 
each particle by a slightly different amount. Here in the 
measurement update we just read offa value from a Gaussian 
function to obtain a likelihood for each particle.  with an offset of +10 and with the s.d. Of the calculated one just now

Perhaps (one of many): 

Giving, mean= 12-0.1v, sd = v/10

4a) Idea: we find the location of the cell by new\_x = x + z \* cos (alpha + theta) (z is distance, alpha + theta is angle to reach cell). For each x, y in the occupancy map we determine their vector c [x y]^T, the distance of c, c\_dist (sqrt (x^2 + y^2)) and find their dot product with the vector represented by the distance z ([new\_x new\_y]^T). We find beta = cos^-1(dot product / z \* c\_dist) and see if this is less than 5 degrees (half width stated). If so then if z - c\_dist < 4 and z - c\_dist > -4 then we update the occupancy map to some value > 0.5, say 0.7. Otherwise we set it to 0.3. If beta > 5 w e do nothing to the value in the occupancy map.

BEAM\_SIGMA = 4

BEAM\_HALF\_WIDTH = 5

def update\_occupancy(x: float, y: float, theta: float, z: float, alpha: float):

for global\_x in range(500):

for global\_y in range(500):

# Position of the cell relative to the robot.

relative\_x = global\_x - x

relative\_y = global\_y - y

beam\_x = z \* math.cos(math.radian(theta + alpha))

beam\_y = z \* math.sin(math.radians(theta + alpha))

if not is\_within\_beam(relative\_x, relative\_y,

beam\_x, beam\_y, math.radians(BEAM\_HALF\_WIDTH)):

continue

cell\_distance = math.sqrt(relative\_x\*\*2 + relative\_y\*\*2)

if abs(z - cell\_distance) < BEAM\_SIGMA:

occupancyMap[global\_x][global\_y] += 5

elif cell\_distance < z - BEAM\_SIGMA:

occupancyMap[global\_x][global\_y] -= 2

def is\_within\_beam(cell\_x, cell\_y, beam\_x, beam\_y, beam\_half\_width):

cell\_dot\_beam = cell\_x \* beam\_x + cell\_y \* beam\_y

cell\_distance = math.sqrt(cell\_x\*\*2 + cell\_y\*\*2)

beam\_distance = math.sqrt(beam\_x\*\*2 + beam\_y\*\*2)

angle\_to\_beam = math.acos(cell\_dot\_beam / (cell\_distance \*

beam\_distance))

return angle\_to\_beam <= beam\_half\_width

b) For (x, y) we know whether a cell around it would have probability 0.3 or 0.7. We could keep track of the number of cells with 0.3 and 0.7 around it and perhaps their location in relation to the current coordinates and use this to later determine our location.

Alternative:

For each angle in the signature you want to calculate, “simulate” the beam going forwards until it would hit an occupied cell (taking the width maybe into account), and store that distance.

Alternative mk2:

Simulate a full sonar rotation but use the existing data in occupancyMap, then do place recognition as normal. If the weight is above a threshold, it counts as an obstacle, and therefore you use the distance to the nearest obstacle at each degree of simulated sonar scan as the measurement to store. Do this every 10 cm along x and y to generate a set to store in a file.