**lNote: there's a second separate robotics folder in addition to this one! It's in the MSc Computing folder (no clue why there's two)**

1a.

The encoder records the angular position of the motor.

In the **Power Control** method there is no communication with the encoder. We simply give a set amount of power to the motor and it will run at the related speed. If the resistance is increased the wheel will slow down since it will require more force to turn the w heel. The velocity of the wheel is proportional to the input Power and resistance.

In the **Position Control** method, we evaluate the current angle of the wheel (the encoder gives us this value) and we then turn the wheel (presumably using a PID controller) until the encoder angle matches the argument of SetMotorPosition. We want to slow down the wheel velocity as we near the target position for accuracy. If extra resistance is added the wheel will still end at the same rotation but will take longer to reach that point.

In **Velocity Mode** the wheel spins at a fixed speed based on the input argument. If the resistance increases the wheel, we continue rotating at the same speed by increasing the input power. It uses the encoder to detect the current wheel angle for the angle per second measurement.

1b.

e - represents the error in relation to the robot's position after a period of motion in a straight line compared to its expected position.

If this is doubled then there will be a small increase in both the x,y directions at each point however the robot will still make it to the end for every point.

(Wouldn’t the increases in x and y directions depend on which direction the robot is traveling? So in the first 3 movements, the particle cloud would expand vertically, then horizontally for the next 2, then vertically again for the remainder)

f - represents the error in relation to the robot's rotation after a period of motion in a straight line compared to its expected rotation.

If this is doubled then there will be a swing in the y direction at every step (if x is the robot facing the next target). This error is generally not too big so the robot will still make it to the end for every point.

g - represents the error in relation to the robot's rotation after a period of pure rotation compared to its expected rotation.

If this is doubled then there will be a bigger swing in the y direction at every step (if x is the robot facing the next target). Because this leads to the robot facing in the wrong direction there is a good chance at least some of the particles will hit walls.

1c.-

Lua:

/\* Assume particles is a table of 100 objects with x, y, theta and weight attributes.

/\* Assume sigma\_pos and sigma\_turn are standard deviations associated with forward motion and turning motion. Assume D is the distance set for one robot step forward.

/\* Assume robotRadius is the robot radius, perhaps set in sysCall\_init().

/\* Assume stop() is a system call or function that causes robot processes to exit to stop motion.

function updateParticles()

probCollision = 0.0

for i=1,100 do

e = random.gauss(0, sigma\_pos ^ 2)

f = random.gauss(0, sigma\_turn ^ 2)

particles[i].x = particles[i].x + (D+e)\*cos(particles[i].theta))

particles[i].y = particles[i].y + (D+e)\*sin(particles[i].theta))

particles[i].theta = particles[i].theta + f

if (particles[i].y > 0.5 - robotRadius or particles[i].y < -0.5 + robotRadius) then

probCollision += 0.01

end

end

if (probCollision >= 0.05) then

stop()

end

end

Python:

* DriveForwardD()
* Count = 0
* While (Count < 5)
  + Count = 0
  + Foreach (var n in N)
    - n\_e = random.gauss(0, e)
    - n\_f = random.gauss(0, f)
    - n.x += (D + n\_e) \* cos(n.theta)
    - n.y += (D + n\_e) \* sin(n.theta)
    - 、
    - n.theta = n.theta + n\_f
    - If (n.y >= 1 || n.y <= -1)
      * Count += 1

2a.

Lua:

function GetCameraLikelihood(particle,observations)

numberOfObservations = observations.length

likelihood = 1

for i = 1, numberOfObservations, 1 do

-- distance = (x^2 + y^2)^0.5

distanceX = particle.x - observations[numberOfObservations].x

distanceY = particle.y - observations[numberOfObservations].y

distance = math.pow(math.pow(distanceX,2)+math.pow(distanceY,2),0.5)

-- atan2 handles the quadrant issue that comes up

-- I think ????? Please edit/add if it’s wrong :3

angle = math.atan2(distanceY/distanceX)

likelihood = likelihood \* math.pow(e,-(math.pow(angle,2) /2\*math.pow(math.rad(3),2)))

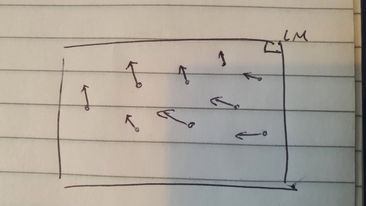
end

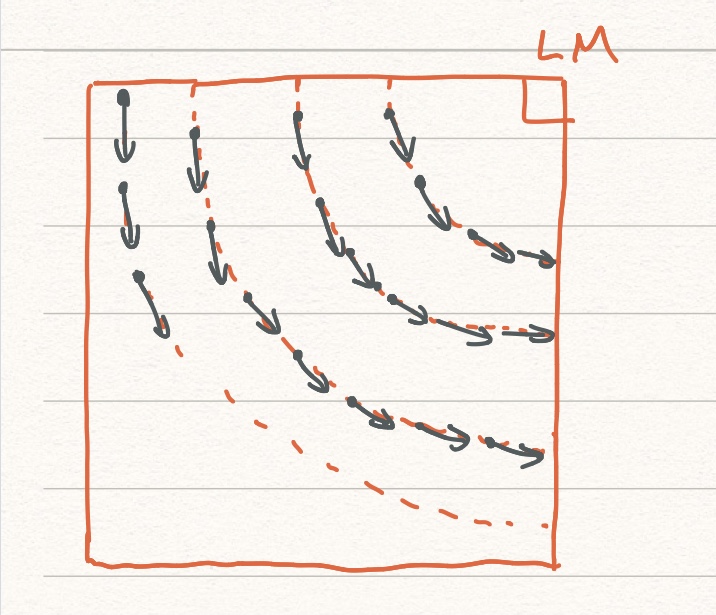
end

Python:

* Likelihood = 1
* foreach (var in observations)
  + landmark = landmarks[observation.id]
  + dist\_x = landmark.x - particle.x
  + dist\_y = landmark.y - particle.y
  + angle = tan -1(dist\_y/dist\_x)
  + If dist\_x < 0 then angle + 180
  + If angle < 0 then angle + 360
  + particle\_to\_landmark = particle.theta - angle
  + ptl\_mod = Modulus between –pi/2 and pi/2 (angle in relation to particle) (left to right)
  + alpha\_angle\_compared = (observation.alpha - ptl\_mod) mod 2pi
  + if alpha\_angle\_compared > 180
    - alpha\_angle\_compared = 360 - alpha\_angle\_compared
  + Likelihood \*= e^((- alpha\_angle\_compared)^2/2\*0.001^2)
* Return likelihood

2b.



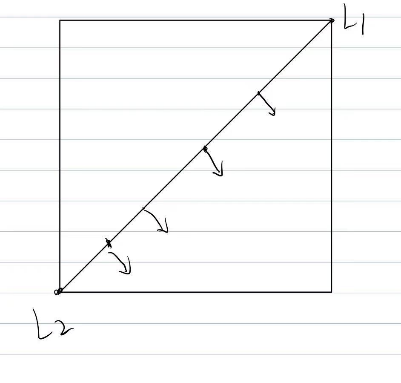


2c.

2

(Might need 3 observation

s in this situation?



You definitely need 3 landmarks, 1 observation from each landmark.

)

2d.

~~Using the observation estimations, we can estimate the rotation of the robot at different locations by weighing the particle likelihoods. We can compare the answers we get to the compass readings to determine its accuracy.~~

Alternative (this is what he says in the lecture): place the robot in different orientations, get the reading from the compass and compare the reading to the ground truth orientation. Finally, calculate the std dev from these readings (so we can use it in the likelihood function).

For GetCompassLikelihood We can compare the reading the compass gives with the particle rotation.

2e.

* + Motion Prediction based on proprioceptive sensors.
  + Measurement update based on outward-looking sensors
  + Normalization
  + Resampling

We use the sensors for the measurement update. We can sum the measurements from both sensors and then normalize them to obtain good particle choices.

We can obtain sensor measurements for both the compass and camera sensor, calculate their likelihoods and **multiply** them together (assuming the measurements are independent).

3a.

* for i in range (500),
  + for j in range (500)
    - c\_x = i – x
    - c\_y = j – y
    - dist = norm(c\_x,c\_y)
    - If (dist > z + 4) continue
    - z\_x = z\*cos(theta+alpha)
    - z\_y = z\*sin(theta+alpha)
    - Beta = cos-1 ((z\_x\*c\_x + c\_y\*z\_y)/(z\* dist)
    - k
    - If (Beta > 5) continue
    - If (dist < z – 4) increase log odds # occupancy\_map[x][y] += 5
    - Else decrease log odds # occupancy\_map[x][y] -=2

3b.

* Get x, y, theta position
* Retrieve value of the occupancy map cell that is 30cm in front of the robot
* If it states, there is no obstacle then move forward and loop
* If there is an obstacle rotate to the right and loop
* In practice, you would need to do this for a line the size of the robot. Need to ensure neither side of the robot collides.