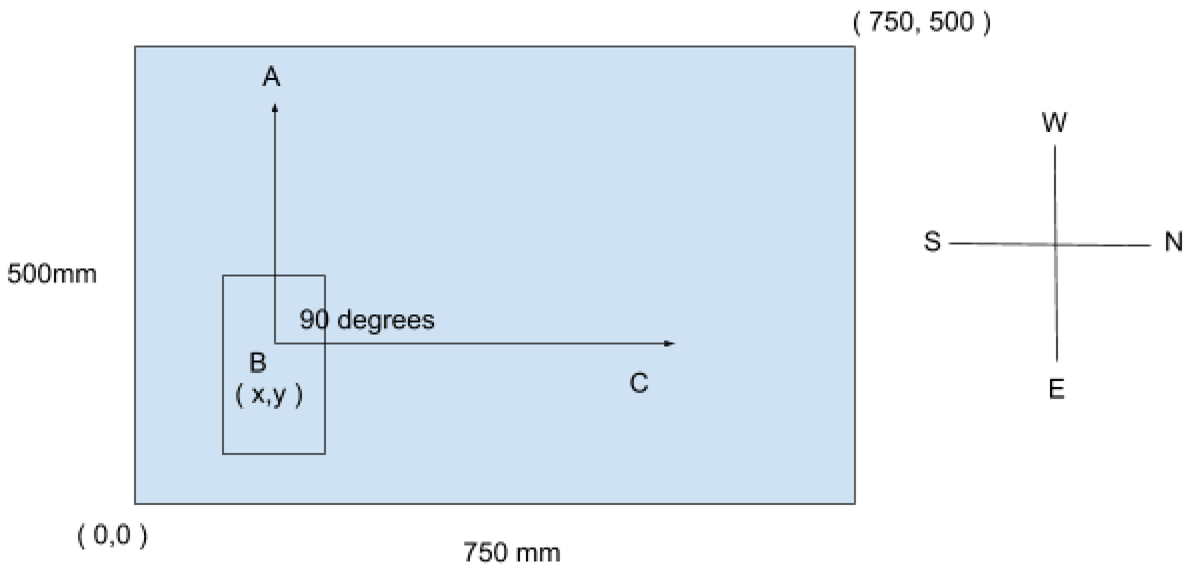
PART 2 MATHEMATICAL SETUP

Question 2(a)

We have defined our state to be a vector of length 4. The vector is [X Y θ ω].

```
class State:
  """We create an object called State which has x coordinate ,
  y coordinate, angle with respect to magnetic north, and angular velocity"""
  def __init__(self, x, y, angle, omega):
      self.x = x
      self.y = y
      self.angle = np.radians(angle)
      self.omega = omega
```

Here X and Y represent the position of the robot in our 750mm X 500mm environment. The angle θ represents the bearing of the robot with respect to the magnetic north. The angular velocity is represented as ω. The figure below illustrates this:



In this picture, the point B is the location of the robot, represented by (x, y). AB is the heading of the robot. BC is the magnetic north direction. So in this case, θ is 90°.

We have two wheels, left and right, whose angular speeds are represented by w_L and w_R respectively. As provided in the Lab, they both have a maximum speed of 60RPM or 1 rotation per second or in terms of radians/second, we get 2π radians/sec.

The velocity of a wheel v is given by the following formula:

$$V = \omega R$$

Here w is the angular speed of the wheel and R is the radius of the wheel in question.
Using this formula in our case, we obtain the following equations for the speeds of the right wheel and the left wheel.

$$V_R = \omega_R r$$

$$V_L = \omega_L r$$

Here R being the radius of the wheel which is 25mm. The velocity of the robot is then calculated using the following equation:

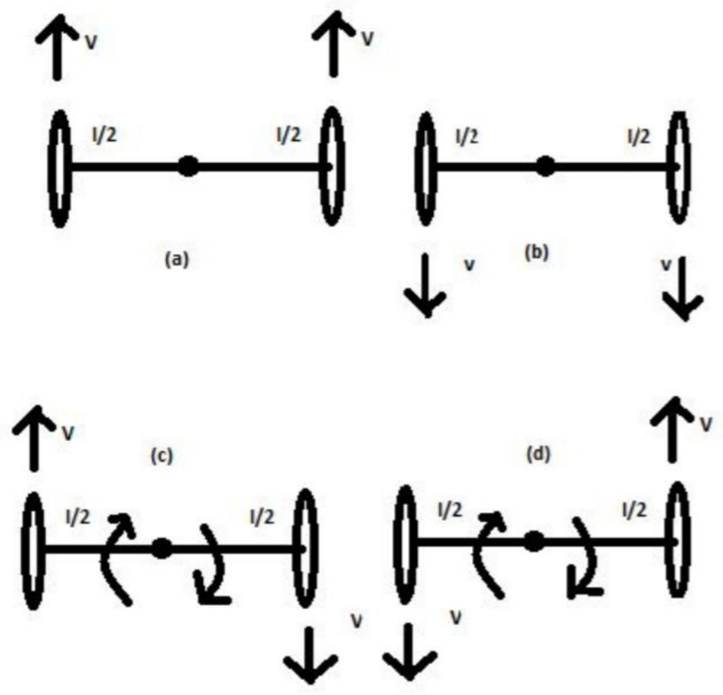
$$V_{robot} = \frac{V_L + V_R}{2}$$

The change is the robot's heading denoted by the variable omega_robot in our code is then given by the following equation:

$$\omega_{robot} = \frac{V_R - V_L}{length}$$

The length of the robot is 90mm.

We will need to add omega_robot to our robot's current bearing to get the new updated bearing of our robot.



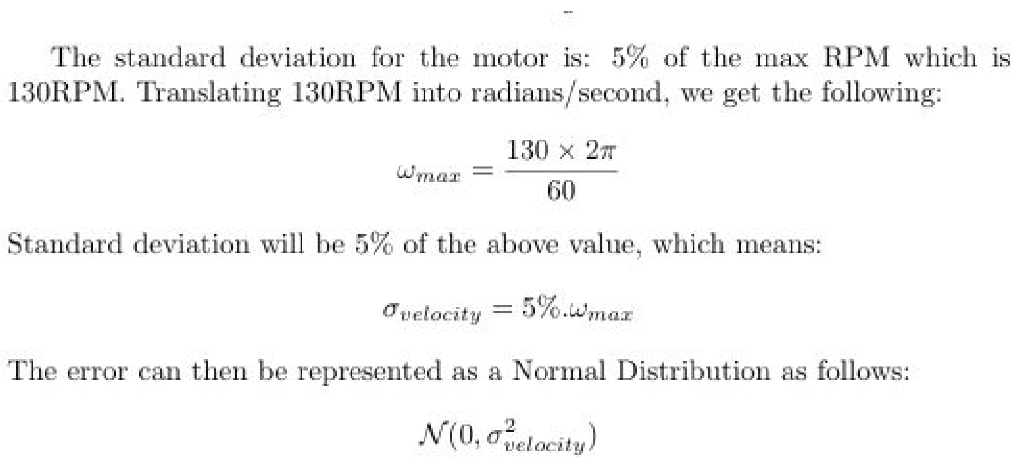
The **sensor and actuator model** involves calculating the distance of the robot's current position from the boundary points which are obtained from *sensor_output_1* and *sensor_output_2.* These functions use trigonometric functions to calculate the boundary points the laser would hit, and it calculates the distance using the function *Hx.* The function *reflection_point* helps us calculate the position of these landmarks on the boundary by dividing the cartesian plane into four quadrants and handling individual cases. The code structure can be obtained on our github link.

```python
def update_state(my_State, dt, my_wheel_angular_velocity):
  """given current state and input returns the next state"""
  angle = normalize_theta(my_State.angle)
  next_State = State(my_State.x, my_State.y, np.degrees(angle), my_State.omega)
  V_wheel_left  = my_wheel_angular_velocity.w_l * robot_radius
  V_wheel_right = my_wheel_angular_velocity.w_r * robot_radius
  #linear velocity and displacement
  v_robot = (V_wheel_left + V_wheel_right) / 2
  d_s = v_robot * dt

  #angular velocity and displacement
  omega_robot = (V_wheel_right - V_wheel_left) / (robot_length)
  d_theta = omega_robot * dt
  angle = normalize_theta(next_State.angle + d_theta/2)
  dx_robot = d_s * np.cos(angle)
  dy_robot = d_s * np.sin(angle)
  next_State.x = next_State.x + dx_robot
  next_State.y = next_State.y + dy_robot
  next_State.angle = angle
  next_State.omega = omega_robot
  return next_State

def normalize_theta(theta):
  theta = theta % (2 * np.pi )
  if theta >= np.pi:
    theta -= 2* np.pi
  return theta
```

Question 2(b)

For this lab, we have modelled all our noises as Gaussian distributions. We shall go over each of the noises and explain the parameters of the distribution.

1.



The standard deviation for the motor is: 5% of the max RPM which is 130RPM. Translating 130RPM into radians/second, we get the following:

$$\omega_{max} = \frac{130 \times 2\pi}{60}$$

Standard deviation will be 5% of the above value, which means:

$$\sigma_{velocity} = 5\%.\omega_{max}$$

The error can then be represented as a Normal Distribution as follows:

$$\mathcal{N}(0, \sigma^2_{velocity})$$

2.

The laser range sensor according to the data sheet has a standard deviation of 1200×0.03. This results in the following Normal Distribution.

$$\sigma_{laser} = 1200 \times 0.03$$

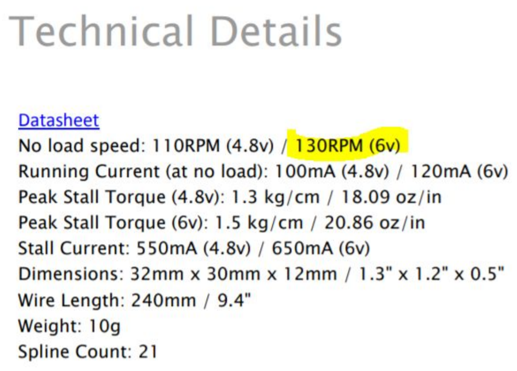
$$\mathcal{N}(0, \sigma_{laser}^2)$$

3.

For the gyro, we get a standard deviation of 0.1 degrees according to the data sheet for the IMU unit. $\sigma_{gyro} = 0.1$ degrees. This would result in a normal distribution as follows:

$$\mathcal{N}(0, \sigma_{gyro}^2)$$

We obtain the noises from the datasheets, for example from the datasheet of FS90R we obtain max rotor speed of 130 rpm and calculate 5% of that.



Technical Details

Datasheet
No load speed: 110RPM (4.8v) / 130RPM (6v)
Running Current (at no load): 100mA (4.8v) / 120mA (6v)
Peak Stall Torque (4.8v): 1.3 kg/cm / 18.09 oz/in
Peak Stall Torque (6v): 1.5 kg/cm / 20.86 oz/in
Stall Current: 550mA (4.8v) / 650mA (6v)
Dimensions: 32mm x 30mm x 12mm / 1.3" x 1.2" x 0.5"
Wire Length: 240mm / 9.4"
Weight: 10g
Spline Count: 21

FS90R Datasheet

Question 2(c)
For this lab we chose to implement an Extended Kalman Filter over an Unscented Kalman Filter. One of the main reasons is computational power. The EKF runs on a single point whereas the UKF uses a collection of points to calculate its estimate. Another reason for choosing the EKF is that in this particular case, the Jacobian is fairly straightforward to calculate. There are some cases where the Jacobian might be harder to calculate, but it was not the case here.

We will now describe the equations of the EKF. The main advantage of the EKF is that the state transitions and the observation models do not have to be linear. The filter however approximates them to be linear.

$$X_t = F(X_{t-1}, u_t) + w_t$$

Here, X_t represents the state at a given time t. F is the state transition matrix, u_t is the input at time t and w is the associated noise that follows a Gaussian distribution.

The Observation can be explained by the following equation:

$$Y_t = H(X_t) + v_t$$

Here Y_t is the observation at time t and v_t is the associated noise that follows a Gaussian distribution.

Our state updates can be visualized as follows:

$$\begin{bmatrix} \overline{\theta}_{t+1} \\ \overline{x}_{t+1} \\ \overline{y}_{t+1} \\ \overline{\omega}_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{\theta}_t + \omega_t \Delta t \\ \hat{x}_t + v_t cos\hat{\theta}_t \Delta t \\ \hat{y}_t + v_t sin\hat{\theta}_t \Delta t \\ \hat{\omega}_t \end{bmatrix}$$

Our observation model can be described in the following equation:

$$\begin{bmatrix} \phi_t \\ r_{1t} \\ r_{2t} \end{bmatrix} \approx \begin{bmatrix} tan^{-1}(\frac{y_t - \lambda_{y1}}{x_t - \lambda_{x1}}) - \theta_t \\ \sqrt{(x_t - \lambda_{x_1})^2 + (y_t - \lambda_{y_1})^2} \\ \sqrt{(x_t - \lambda_{x_2})^2 + (y_t - \lambda_{y_2})^2} \end{bmatrix}$$

PART 3 EVALUATION

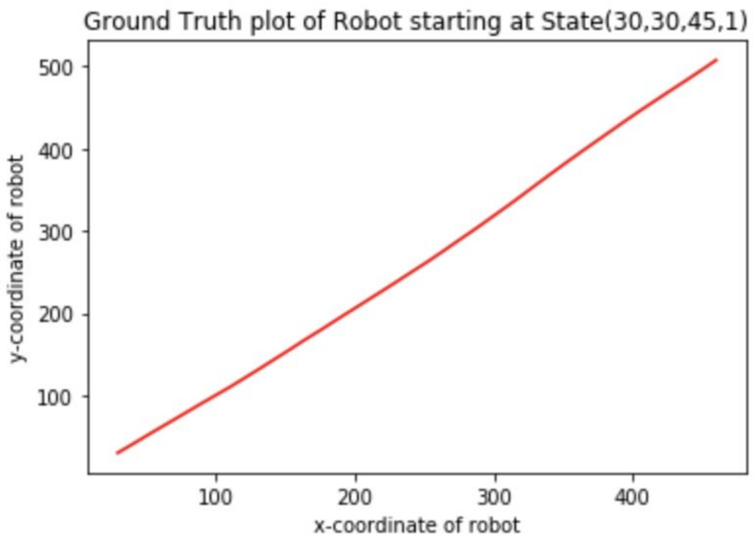
Question 3(a)

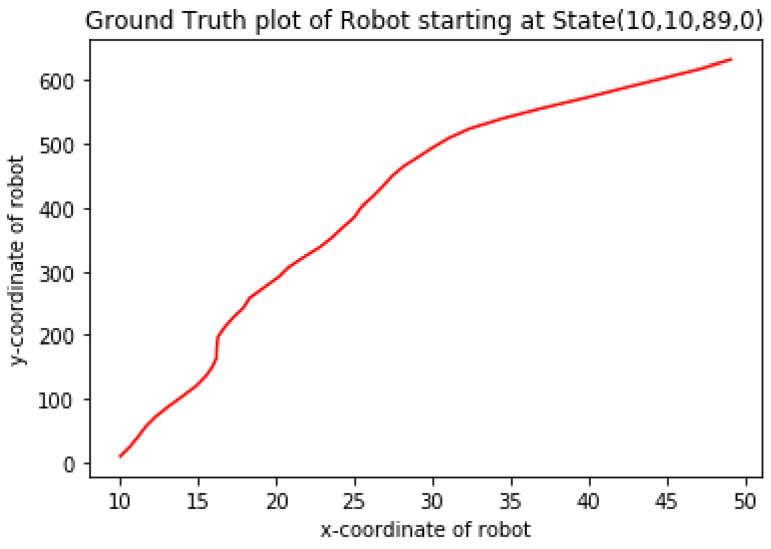
The trajectories that are plotted below is of the ground truth.
Ground truth is found by passing the state into our noise_model.
Note the only noise that ground truth has is from the slip in the wheels (omega_std).
Noise in the sensors (range_std) or noise from gyro (angle_std) is not incorporated.

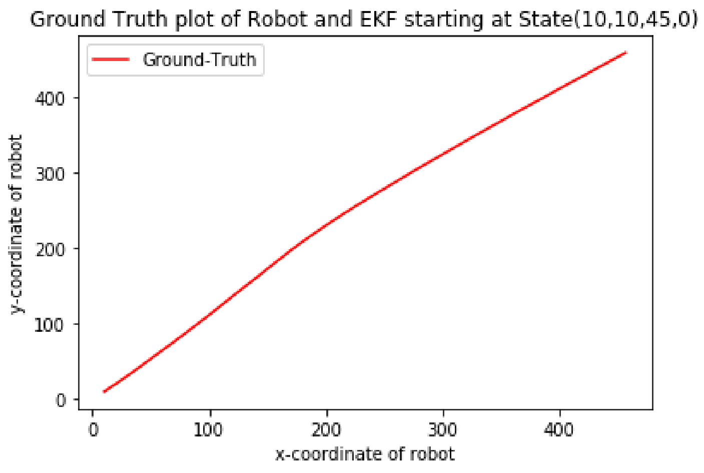
We implemented in the ground truth in our simulate_path_noise_with_EKF function.

```
#Ground Truth
front_truth, right_truth, new_w_truth, gtruth_state = noise_model(gtruth_state, range_std, angle_std, omega_std, new_w, False)
gtruth_state = update_state(gtruth_state, dt, new_w_truth)
```

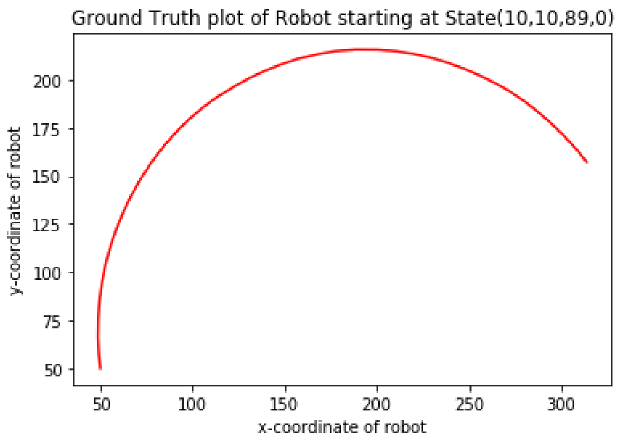
Below are the ground truth trajectories for different initial states:


Ground Truth plot of Robot starting at State(30,30,45,1)
y-coordinate of robot
x-coordinate of robot
500
400
300
200
100
100
200
300
400


Ground Truth plot of Robot starting at State(10,10,89,0)
y-coordinate of robot
x-coordinate of robot
600
500
400
300
200
100
0
10
15
20
25
30
35
40
45
50

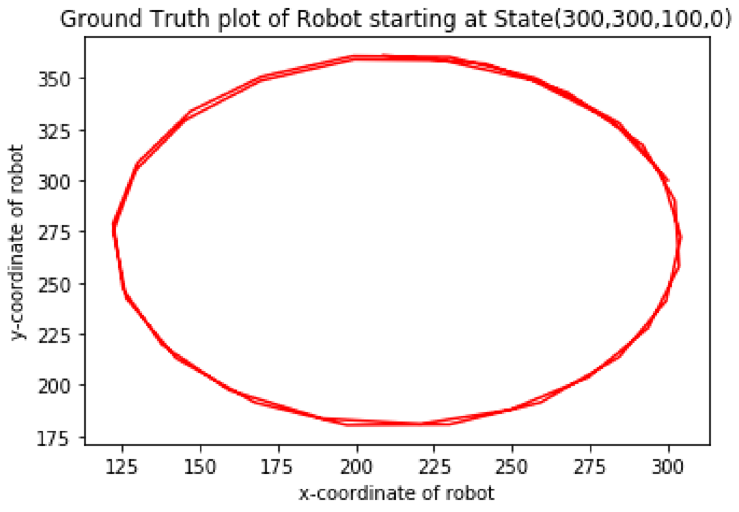


Curved paths are generated by changing the omega of the wheels:
Omega_right of the wheel = $\omega r/2$



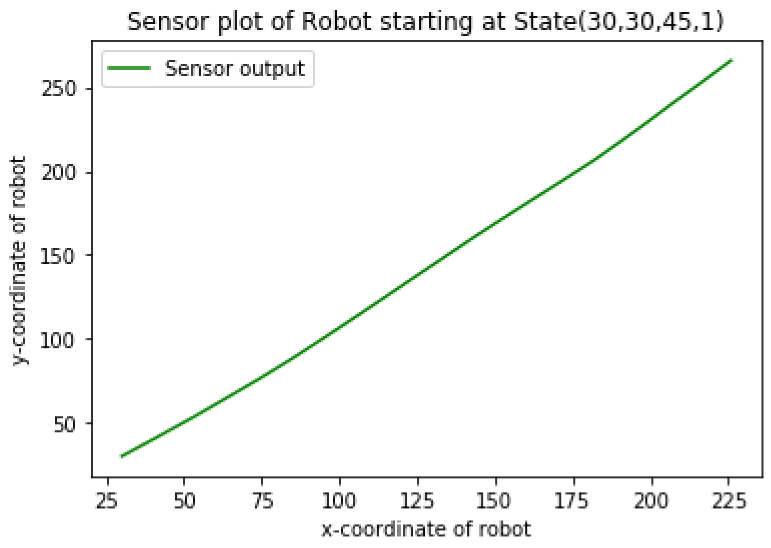
Curved paths below is generated by changing the omega of the wheels:

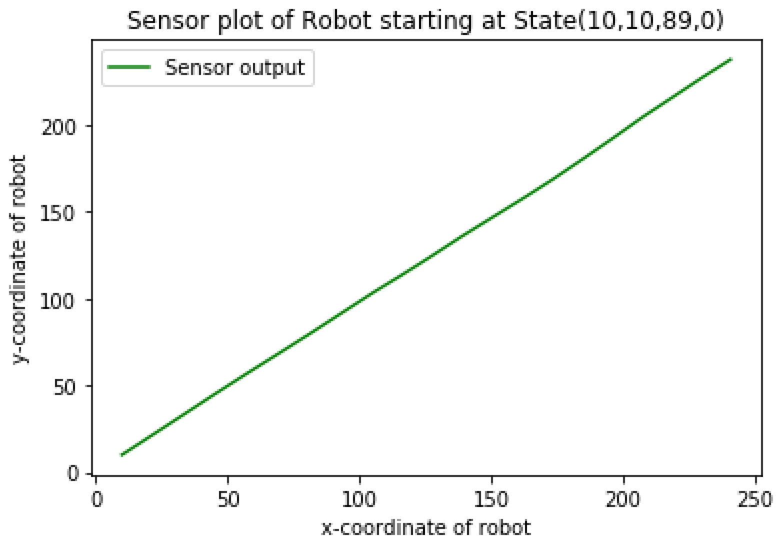
```
self.w_l = w_l/9
self.w_r = w_r*2
```



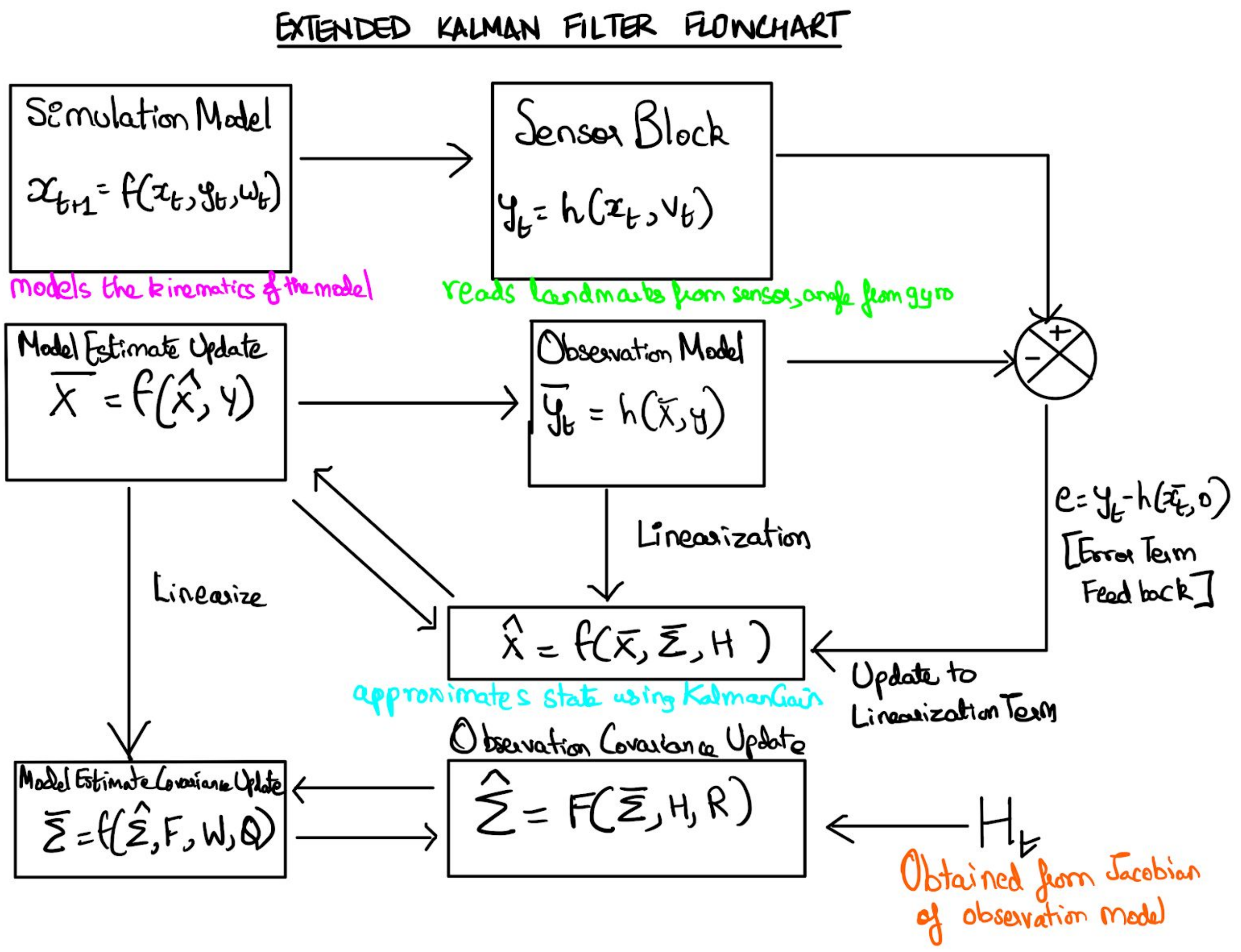
Question 3(b)

The trajectories that are plotted below is output from our sensors.
This will include the noise from all:
Noise in the sensors (range_std)
Noise from gyro (angle_std)
Noise from the wheel slip (omega_std)





Question 3(c)
Implementation of the Extended Kalman Filter (EKF)
We used the below flowchart to implement our EFK model:

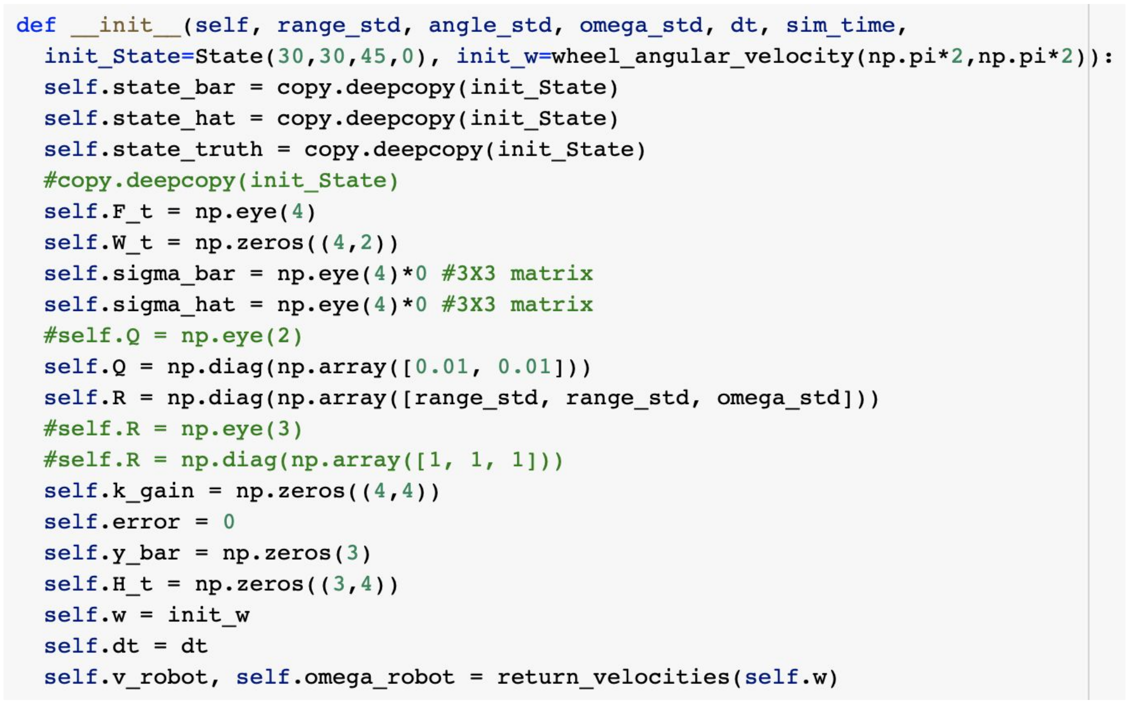

EXTENDED KALMAN FILTER FLOWCHART
Simulation Model
$x_{t+1} = f(x_t, y_t, \omega_t)$
models the kinematics of the model
Sensor Block
$y_t = h(x_t, v_t)$
reads landmarks from sensor, angle from gyro
Model Estimate Update
$\bar{X} = f(\hat{x}, y)$
Observation Model
$\bar{y}_t = h(\bar{x}, y)$
+
−
$e = y_t - h(\bar{x}_t, 0)$
[Error Term Feedback]
Linearize
Linearization
$\hat{x} = f(\bar{x}, \bar{\Sigma}, H)$
approximates state using KalmanGain
Update to Linearization Term
Observation Covariance Update
Model Estimate Covariance Update
$\bar{\Sigma} = f(\hat{\Sigma}, F, W, Q)$
$\hat{\Sigma} = F(\bar{\Sigma}, H, R)$
H_t
Obtained from Jacobian of observation model

Our state is 4-dimensional with θ, x-position, y-position, and ω.
We created a State class with the four attributes: θ, x-position, y-position and ω.
We will be creating states calling the constructor for the class State:

```python
class State:
  """We create an object called State which has x coordinate ,
  y coordinate, angle with respect to magnetic north, and angular velocity"""
  def __init__(self, x, y, angle, omega):
      self.x = x
      self.y = y
      self.angle = np.radians(angle)
      self.omega = omega
```

Notations used:
We used the **“bar”** notation for all time propagation updates
And we have used the **“hat”** notation for all the observation model updates
Noise from sensors: range_std
Noise from gyro: angle_std
Noise from wheel slip: omega_std

EKF initialization Step:
Initializes our ekf object with the following constructor
Initial $\omega = \omega_l = \omega_r = 2\pi$



```
def __init__(self, range_std, angle_std, omega_std, dt, sim_time,
  init_State=State(30,30,45,0), init_w=wheel_angular_velocity(np.pi*2,np.pi*2)):
  self.state_bar = copy.deepcopy(init_State)
  self.state_hat = copy.deepcopy(init_State)
  self.state_truth = copy.deepcopy(init_State)
  #copy.deepcopy(init_State)
  self.F_t = np.eye(4)
  self.W_t = np.zeros((4,2))
  self.sigma_bar = np.eye(4)*0 #3X3 matrix
  self.sigma_hat = np.eye(4)*0 #3X3 matrix
  #self.Q = np.eye(2)
  self.Q = np.diag(np.array([0.01, 0.01]))
  self.R = np.diag(np.array([range_std, range_std, omega_std]))
  #self.R = np.eye(3)
  #self.R = np.diag(np.array([1, 1, 1]))
  self.k_gain = np.zeros((4,4))
  self.error = 0
  self.y_bar = np.zeros(3)
  self.H_t = np.zeros((3,4))
  self.w = init_w
  self.dt = dt
  self.v_robot, self.omega_robot = return_velocities(self.w)
```

Time Update Model:
The equations for time update model are given below:
We update the state bar (θ_bar, x_bar, y_bar, omega_bar) by passing the state hat updates (from observation model) using the following equations. Note that we normalize our θ’s to ensure the θ is between -π and π.

$$\begin{bmatrix} \overline{\theta}_{t+1} \\ \overline{x}_{t+1} \\ \overline{y}_{t+1} \\ \overline{\omega}_{t+1} \end{bmatrix} = \begin{bmatrix} \hat{\theta}_t + \omega_t \Delta t \\ \hat{x}_t + v_t cos\hat{\theta}_t \Delta t \\ \hat{y}_t + v_t sin\hat{\theta}_t \Delta t \\ \hat{\omega}_t \end{bmatrix}$$

```python
def time_update(self):
 #updates self.state_bar using self.state_hat from observation model
 theta_t_next = self.state_hat.angle + self.omega_robot * self.dt
 theta_t_next = normalize_theta(theta_t_next)
 print(theta_t_next)
 x_t_next = self.state_hat.x + (self.v_robot * np.cos(self.state_hat.angle) * self.dt)
 y_t_next = self.state_hat.y + (self.v_robot * np.sin(self.state_hat.angle) * self.dt)
 omega_t_next = self.state_hat.omega
 self.state_bar = State(x_t_next, y_t_next, np.degrees(theta_t_next), omega_t_next)
```

Noise propagation by linearization:

$$\begin{bmatrix}\widetilde{\theta}_{t+1}\\ \widetilde{x}_{t+1}\\ \widetilde{y}_{t+1}\\ \widetilde{\omega}_{t+1}\end{bmatrix} \approx \begin{bmatrix} 1 & 0 & 0 & \Delta t\\ -v_t sin\hat{\theta}_t\Delta t & 1 & 0 & 0\\ v_t cos\hat{\theta}_t\Delta t & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}\widetilde{\theta}_t\\ \widetilde{x}_t\\ \widetilde{y}_t\\ \widetilde{\omega}_t\end{bmatrix} + \begin{bmatrix}\Delta t & 0\\ 0 & cos\hat{\theta}_t\Delta t\\ 0 & sin\hat{\theta}_t\Delta t\\ 1 & 0\end{bmatrix}\begin{bmatrix}\omega_{\omega,t}\\ \omega_{v,t}\end{bmatrix}$$

This equals to the below:

$$F_t\begin{bmatrix}\widetilde{\theta}_t\\ \widetilde{x}_t\\ \widetilde{y}_t\\ \widetilde{\omega}_t\end{bmatrix} + W_t\begin{bmatrix}\omega_{\omega,t}\\ \omega_{v,t}\end{bmatrix}$$

We are able to obtain F_t and W_t matrix using the below time_linearize function:

```python
def time_linearize(self):
  #updates self.F_t, self.W_t
  self.F_t[1][0] = - self.v_robot * np.sin(self.state_hat.angle) * dt
  self.F_t[2][0] = self.v_robot * np.cos(self.state_hat.angle) * dt
  self.F_t[0][3] = dt
  self.W_t[0][0] =  dt
  self.W_t[1][1] = np.cos(self.state_hat.angle) * dt
  self.W_t[2][1] = np.sin(self.state_hat.angle) * dt
  self.W_t[3][0] = 1
```

Next is the covariance update step:
Using the sigma hat from observation model we are updating our sigma bar.

$$\overline{\Sigma}_{t+1} = F_t\hat{\Sigma}_t F_t^T + W_t Q W_t^T$$

We are able to obtain sigma_bar matrix using the below covariance_update function:

```python
def covariance_update(self):
  #updates self.sigma_bar from self.sigma_bar
  sigma_t_next_1 = np.matmul(np.matmul(self.F_t, self.sigma_hat), self.F_t.transpose())
  sigma_t_next_2 = np.matmul(np.matmul(self.W_t, self.Q), self.W_t.transpose())
  self.sigma_bar = sigma_t_next_1 + sigma_t_next_2
```

System Observation Model:

The equations for observation model are given below:

The bearing and range observation model can be expressed as below:

The lambas are the landmarks that are found by using the sensor outputs. They are where the laser range finder hits the wall (intersection of the line from the sensor range finder with the boundary). Our sensor_output_1 function gives the landmark point 1 $(\lambda_{x1}, \lambda_{y1})$ and our sensor_output_2 function gives the landmark point 2 $(\lambda_{x2}, \lambda_{y2})$.

$$\begin{bmatrix} \phi_t \\ r_{1t} \\ r_{2t} \end{bmatrix} \approx \begin{bmatrix} tan^{-1}(\frac{y_t - \lambda_{y1}}{x_t - \lambda_{x1}}) - \theta_t \\ \sqrt{(x_t - \lambda_{x_1})^2 + (y_t - \lambda_{y_1})^2} \\ \sqrt{(x_t - \lambda_{x_2})^2 + (y_t - \lambda_{y_2})^2} \end{bmatrix}$$

Observation Noise approximation:

$$\begin{bmatrix} \widetilde{\phi}_t \\ \widetilde{r}_{1t} \\ \widetilde{r}_{2t} \end{bmatrix} \approx \begin{bmatrix} -1 & -\frac{\overline{y}_t - \lambda_{y1}}{r_{1t}^2} & \frac{\overline{x}_t - \lambda_{x1}}{r_{1t}^2} & 0 \\ 0 & \frac{\overline{x}_t - \lambda_{x1}}{r_{1t}} & \frac{\overline{y}_t - \lambda_{y1}}{r_{1t}} & 0 \\ 0 & \frac{\overline{x}_t - \lambda_{x2}}{r_{2t}} & \frac{\overline{y}_t - \lambda_{y2}}{r_{2t}} & 0 \end{bmatrix} \begin{bmatrix} \widetilde{\theta}_t \\ \widetilde{x}_t \\ \widetilde{y}_t \\ \widetilde{\omega}_t \end{bmatrix} + \begin{bmatrix} v_{\phi,t} \\ v_{r,t} \end{bmatrix}$$

This equals to the below:

$$H_t \begin{bmatrix} \widetilde{\theta}_t \\ \widetilde{x}_t \\ \widetilde{y}_t \\ \widetilde{\omega}_t \end{bmatrix} + \begin{bmatrix} v_{\phi,t} \\ v_{r,t} \end{bmatrix}$$

Using the function observation_update, we can obtain Ht matrix and the y_bar:

Note: y_bar is the output of our observation model which is an array of 3 elements: front_dist, right_dist and θ. These are the R1 is the landmark distance intersection with boundary from sensor1, R2 is the landmark distance intersection with boundary from sensor2 and θ. This y_bar will be eventually be used in our error calculation function.

```
def observation_update(self):
  #updates y_bar which is from the obs model using self.state_bar
  front_dist, right_dist, new_w, self.state_bar = noise_model(self.state_bar, range_std,angle_std,omega_std, self.w, False)
  #front_dist, right_dist, theta = Hx(self.state_bar)
  [x1, y1] = sensor_output_1(self.state_bar, map_length, map_width) #gives front_dist
  [x2, y2] = sensor_output_2(self.state_bar, map_length, map_width) #gives right_dist
  self.y_bar = np.array([front_dist, right_dist, self.state_bar.angle]) #observation model
  # row0: angle from gyro
  self.H_t[0][0] = -1
  self.H_t[0][1] = -(self.state_bar.y - y1)/front_dist**2
  self.H_t[0][2] = (self.state_bar.x - x1)/front_dist**2
  # row1: r from sensor1
  self.H_t[1][0] = 0
  self.H_t[1][1] = (self.state_bar.x - x1)/front_dist
  self.H_t[1][2] = (self.state_bar.y - y1)/front_dist
  # row2: r from sensor2
  self.H_t[2][0] = 0
  self.H_t[2][1] = (self.state_bar.x - x2)/right_dist
  self.H_t[2][2] = (self.state_bar.y - y2)/right_dist
```

Kalman Gain:

$$K_t = \overline{\Sigma}_{t^-} H^\top \left(H \overline{\Sigma}_{t^-} H^\top + R \right)^{-1}$$

The function kalman_gain calculates and updates the k_gain:

```
def kalman_gain(self):
  #updates self.k_gain using self.sigma_bar
  K_1 = np.linalg.inv(self.H_t.dot(self.sigma_bar).dot(self.H_t.transpose()) + self.R )
  self.k_gain = np.dot(self.sigma_bar, self.H_t.transpose()).dot(K_1)
```

Next is the sigma hat covariance matrix update using the below equation:

$$\Sigma_{t^+} = \left(I - K_t H \right) \Sigma_{t^-}$$

Note that sigma_hat is updated using sigma_bar and kalman gain

```
def obs_covariance_update(self):
  #updates self.sigma_hat using self.sigma_bar
  self.sigma_hat = ( np.eye(4) - self.k_gain.dot(self.H_t) ).dot(self.sigma_bar)
```

<u>ASIDE:</u>
We made a function called noise_model that adds noise:

```python
# Including Noise
def noise_model(my_State, range_std, angle_std,omega_std,
                my_wheel_angular_velocity, kalman_flag):
    """ Return the sensor output with simulated noisy"""
    [front_distance,right_distance, theta] = Hx(my_State)
    front_distance += np.random.normal(0, range_std)
    right_distance += np.random.normal(0, range_std)
    w_noisy_l = my_wheel_angular_velocity.w_l + np.random.normal(0, omega_std)
    w_noisy_r = my_wheel_angular_velocity.w_r + np.random.normal(0, omega_std)
    new_w = wheel_angular_velocity(w_noisy_l, w_noisy_r)
    #print("The angle is: " + str(my_State.angle+ randn() *  angle_std))
    if kalman_flag:
      theta += np.random.normal(0, angle_std)
      theta = normalize_theta(theta)
    new_State = State(my_State.x, my_State.y, np.degrees(theta), my_State.omega)
    return front_distance, right_distance, new_w, new_State
```

Note: the noises we have are from our sensors, gyro and the slip in the wheel.
We update the front_distance, right_distance with range_std passed through normal distribution and update ω by passing slip noise (normal dist) and lastly update θ by adding angle_std (noise from gyro).

Error and state hat update:
Using this noise_model function and simulate_path_noise_with_EKF function, we update our state_truth (which is an attribute in our EKF class) -- it is calculated at each time step by finding the R1, R2 and θ from the sensor output. Note: all the noises are included in this -- (i.e. noise from the sensor ranges, noise from gyro and noise from the wheel slip). To calculate the error we subtract the y_bar from this as shown below in the error_calc function.

Next we calculated the error using the error_calc function:

```python
def error_calc(self):
  #updates self.error
  #front_dist_truth, right_dist_truth, theta = Hx(self.state_truth)
  front_dist_truth, right_dist_truth, new_w, new_State_truth = noise_model(self.state_truth, range_std,
                                                                           angle_std, omega_std, self.w,
                                                                           kalman_flag=True)
  print("Printing from within error_calc function")
  print(np.array([front_dist_truth, right_dist_truth, new_State_truth.angle]))
  print(self.y_bar)
  error = np.array([front_dist_truth, right_dist_truth, new_State_truth.angle]) - self.y_bar
  error[2] = normalize_theta(error[2])
  self.error = error
```

Using the error and state bar we finally update the state hat.

The equation we are using is as follows where the error is : $(y_t - H\bar{x}_{t-})$ and found using our error_calc function.

$$\hat{x}_{t+} = \bar{x}_{t-} + K_t\left(y_t - H\bar{x}_{t-}\right)$$

To obtain the state_hat we use the following function called state_hat_update:

```
def state_hat_update(self):
  #updates self.state_hat using self.state_bar
  kalman_update = self.k_gain.dot(self.error)
  #self.state_hat = self.state_bar + kalman_update
  self.state_hat.x = self.state_bar.x + kalman_update[1]
  self.state_hat.y = self.state_bar.y + kalman_update[2]
  self.state_hat.angle = normalize_theta(self.state_bar.angle + kalman_update[0])
  self.state_hat.omega = self.state_bar.omega + kalman_update[3]
  return self.state_hat
```

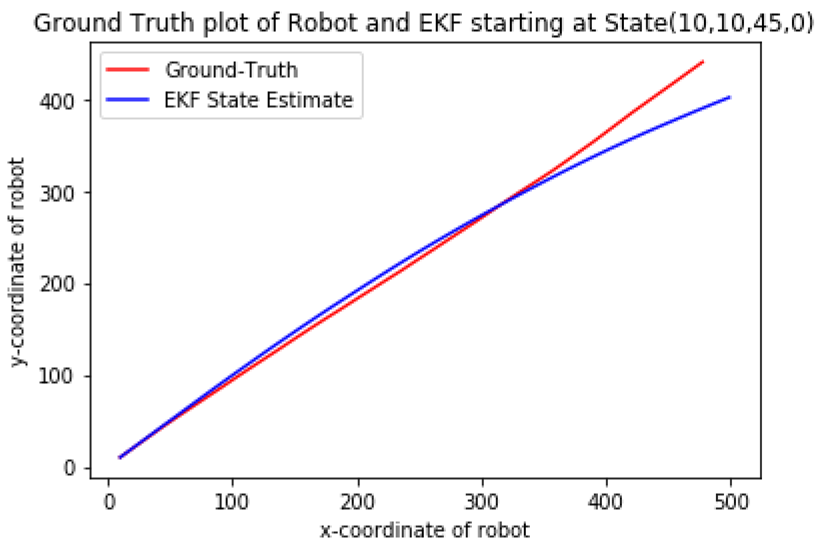
To simulate our EKF easily we made a function called simulate_EFK that calls on all the required functions in order to calculate all the things mentioned above.

```
def simulate_EKF(self):
  self.time_update()
  self.time_linearize()
  self.covariance_update()
  self.observation_update()
  self.kalman_gain()
  self.error_calc()
  self.state_hat_update()
  self.state_hat.print_state()
  self.obs_covariance_update()
```

Lastly to obtain the trajectories, we built a function called simulate_path_noise_with_EKF function, that loops until sim_time and at each time updates the ground truth state and calls the EKF class functions (simulate_EKF function). Using this function, we have plotted the trajectories of our robot and compared the trajectory of ground_state (red) and EKF (blue).

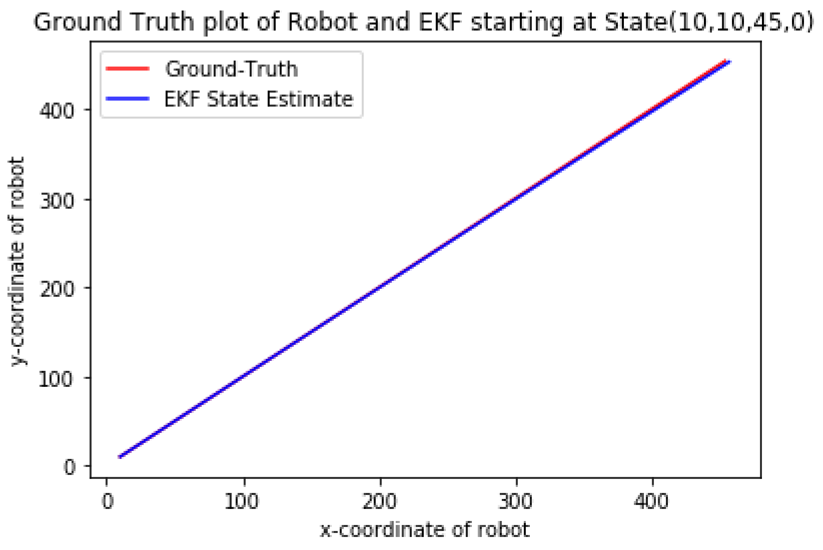
Below are the trajectories with perfect knowledge of the initial state:
Note: we have plotted States at different initial states.
Time step is dt= 0.1seconds.
Notation: In all our plots we have used the notation State(1,2,3,4) where:
1=x-coordinate, 2=y-coordinate, 3=theta, 4=omega

Below is the graph for initial state (10,10,45,0) with simulation time of 15seconds with omega_std set to omega_max*0.05.

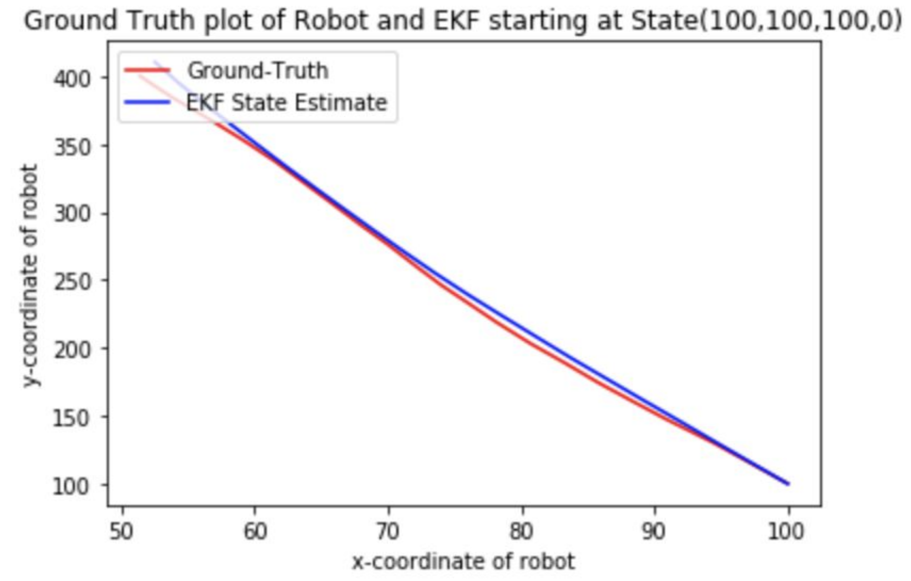


As you can see our EKF is converging to the ground truth.
The small difference that we see (the fact that EKF trajectory is not exactly the same / close to the ground truth) is because of the random noise generated due to the slip in the wheels (omega_std which is set to `omega_max*0.05`). We have not added a bias term to correct this error.

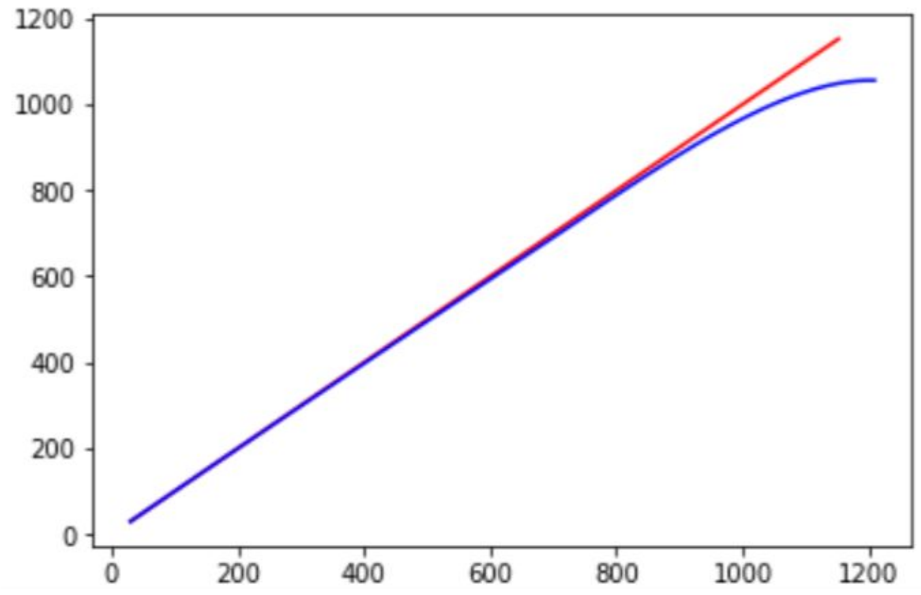
Note in the plot below for the same exact graph but I have set the omega_std noise = 0 and now our trajectories are almost if not exactly the same.



Below is the graph for initial state (100,100,100,0), simulation time of 2 and omega_std= omega_max*0.05

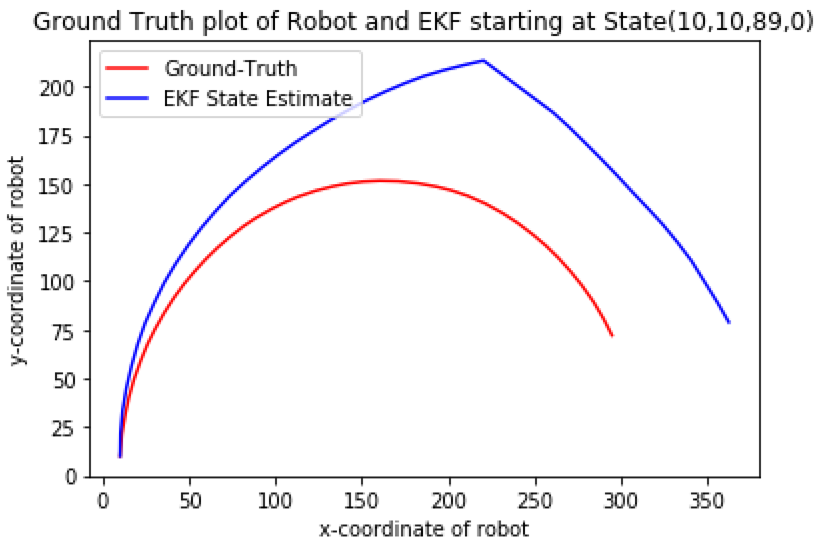


Below is the graph for initial state (30,30,45,0) with simulation time of 10seconds omega_std= omega_max*0.05

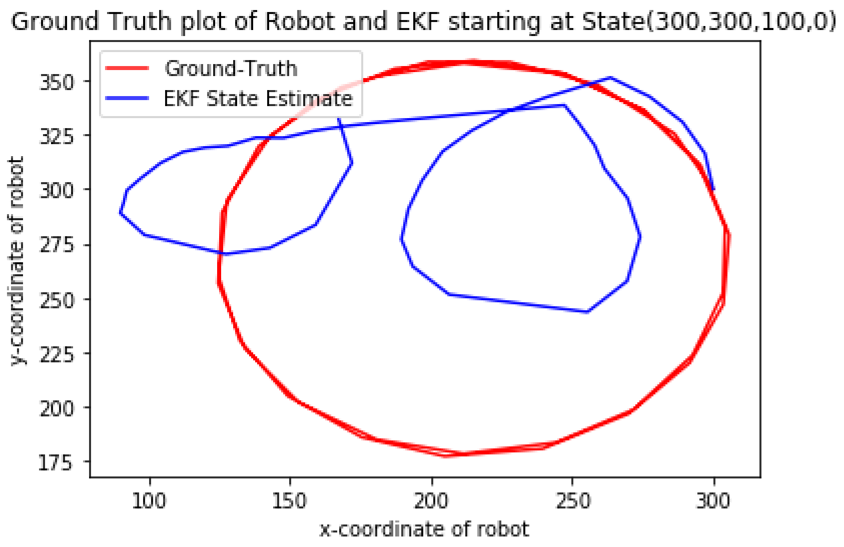


Below are the trajectories of the curved paths:
which we could get by changing the angular velocities of the wheels

Below is the graph for initial state (10,10,89,0) with simulation time of 10seconds omega_std= omega_max*0.05 and omega_right_wheel = omega_right/2

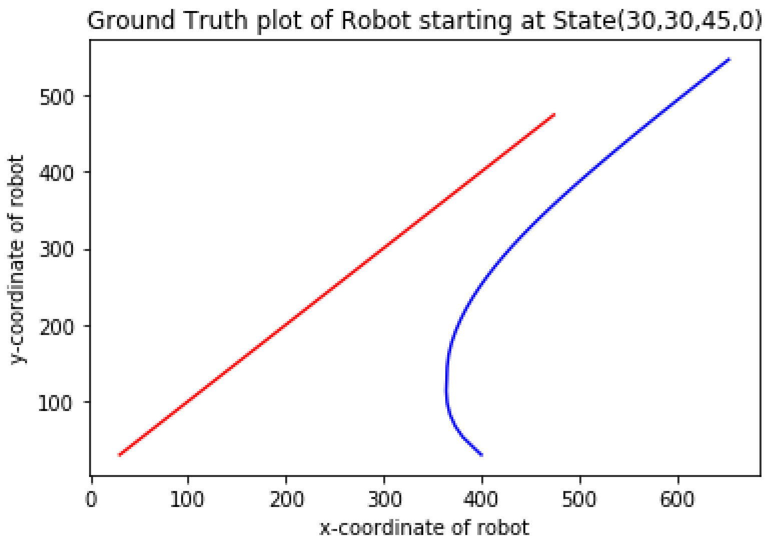


Below is the graph for initial state (10,10,89,0) with simulation time of 10seconds omega_std= omega_max*0.05 and omega of wheels: `self.w_l = w_l/9; self.w_r = w_r*2`



Below is the trajectories with NO knowledge of the initial state:

The following image describes the situation when our Kalman filter is run with initial state unknown. As we can see, our filter is able to get close to the original path. It shows signs of convergence.



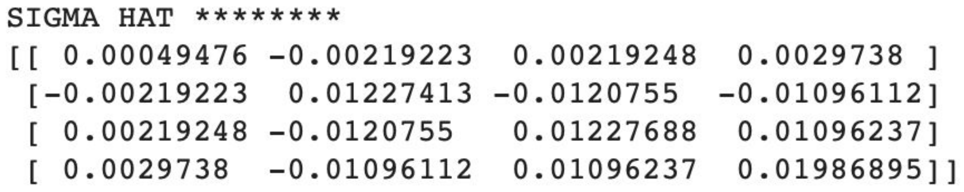
Characterizing the performance of our EKF State Estimator:
Below find the sigma bar and sigma hat of State(10,10,45,0) at three time steps:

TIME STEP 1:

```
 SIGMA BAR ********
[[1.e-04 0.e+00 0.e+00 1.e-03]
 [0.e+00 5.e-05 5.e-05 0.e+00]
 [0.e+00 5.e-05 5.e-05 0.e+00]
 [1.e-03 0.e+00 0.e+00 1.e-02]]
SIGMA HAT ********
[[9.99997222e-05 1.48772960e-29 3.11487023e-29 9.99997222e-04]
 [2.66065806e-29 4.99998533e-05 4.99998533e-05 2.66065806e-28]
 [4.97397955e-29 4.99998533e-05 4.99998533e-05 4.97397955e-28]
 [9.99997222e-04 2.77696814e-28 4.90897975e-28 9.99997222e-03]]
```

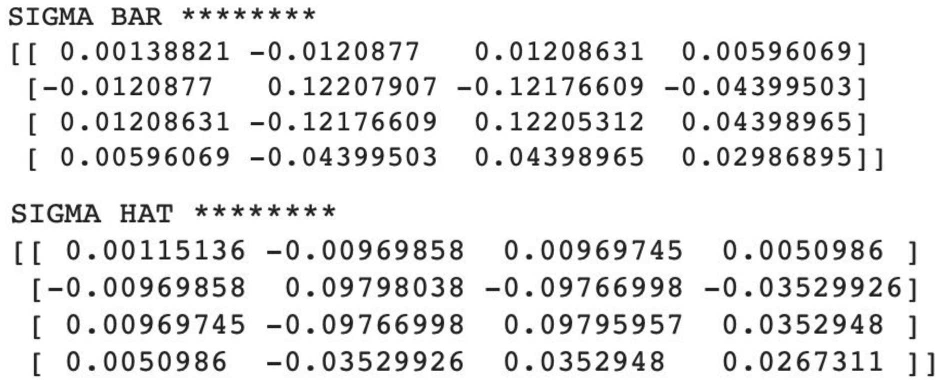
TIME STEP 2:

```
SIGMA BAR ********
[[ 0.0005     -0.00222131  0.00222156  0.00299999]
 [-0.00222131  0.01243558 -0.01223697 -0.01110655]
 [ 0.00222156 -0.01223697  0.01243837  0.01110781]
 [ 0.00299999 -0.01110655  0.01110781  0.01999997]]
```



```
SIGMA HAT ********
[[ 0.00049476 -0.00219223  0.00219248  0.0029738 ]
 [-0.00219223  0.01227413 -0.0120755  -0.01096112]
 [ 0.00219248 -0.0120755   0.01227688  0.01096237]
 [ 0.0029738  -0.01096112  0.01096237  0.01986895]]
```

TIME STEP 3:



```
SIGMA BAR ********
[[ 0.00138821 -0.0120877   0.01208631  0.00596069]
 [-0.0120877   0.12207907 -0.12176609 -0.04399503]
 [ 0.01208631 -0.12176609  0.12205312  0.04398965]
 [ 0.00596069 -0.04399503  0.04398965  0.02986895]]

SIGMA HAT ********
[[ 0.00115136 -0.00969858  0.00969745  0.0050986 ]
 [-0.00969858  0.09798038 -0.09766998 -0.03529926]
 [ 0.00969745 -0.09766998  0.09795957  0.0352948 ]
 [ 0.0050986  -0.03529926  0.0352948   0.0267311 ]]
```

We know that the covariance matrix is influenced by the measurement obtained from the sensors, however the measured values do not affect the covariance matrix, and infact it is the noise related to these sensors that affect the covariance matrix. Hence the covariance matrix gives us a good estimate of the errors in our state, where the diagonal elements represent the variance of our state position and angle. The off-diagonal elements on the other hand depict the correlation between these errors. In short the decrease in uncertainty or values of the diagonal elements of the covariance matrix as shown above for one implementation is a good way to characterize the performance of our Extended Kalman Filter.

Question 3(d)

SInce the gyro does not have a zero mean additive noise it causes some amount of bias in our plots. The best way to tackle this would be to introduce some bias term into our state that would take into account this offset. That way our trajectories would be more accurate. In our code we have not modelled bias and hence we can observe that the errors compound as the simulation time is higher and there exists an offset of our estimated value from the ground truth, since over time the robot drifts, and we cannot make the assumption that the drift noise has mean 0. Had we modelled bias we could include it in the sensor model to reject the bias from the actual sensors.

Question 3(e)

One potential task or use case for our algorithm would be state estimation of the robot or an autonomous vehicle in a plot of land or in a warehouse. Our EKF will be able to estimate where the robot is within this environment. However even though EKF is computationally more efficient, another state-estimation algorithm that we could've used would be the unscented kalman filter (UKF). The UKF does not require computation of a Jacobian and requires just a non-linear state model. There are cases in which the Jacobian cannot be found analytically and hence UKF would be a better choice to make.Also UKF is more accurate than EKF as it does not make a linearization approximation.

References

1) Invensense.com. (2019). [online] Available at: http://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf [Accessed 17 Nov. 2019].
2) [Online]. Available: https://www.st.com/resource/en/datasheet/vl53l0x.pdf.
3) [Online]. Available:https://media.digikey.com/pdf/Data%20Sheets/Adafruit%20PDFs/2442_Web.pdf