Qiong, Zhuyun, Hailin
November 20, 2019

Lab 3 Writeup

EE209AS
Computational Robotics

2 Mathematical setup

2(a). Define the complete system model. You will need to derive and present the mathematical equations describing this robot plant, including the appropriate sensor and actuator models and system dynamics. Note that you will need to amend the state of your robot from the previous lab in order to accommodate this specific collection of sensors. Be sure to clearly define and describe all variables and equations, and produce illustrative diagrams as necessary.

- Define the environment with a map and four walls
  - $L = 750$ mm
  - $W = 500$ mm
- Define the robot dimensions
  - $radius = 25$ mm
  - $length = 100$ mm
  - $width = 90$ mm
  - wheel circumference $C = 2\pi r$
- Define inputs: (w_1, w_2)
  - w_1, w_2: the input angular velocity of the left and right wheel
  - $w_1,\ w_2 \in$ [-ω_{max}, ω_{max}] = [-1, 1], unit: RPS
  - w_1, w_2 are controlled by a PWM signal from the microcontroller, and are independent of each other, because each wheel is powered independently by a continuous rotation servo.
- Define sensor outputs: $(d_1, d_2, \theta, \omega)$
  - d_1: the distance to a wall in a straight line in front of the robot, unit: mm
  - d_2: the distance to a wall in a straight line to the right of the robot, unit: mm
  - θ: an absolute bearing indicating the angle of the robot with respect to magnetic north (direction of +X) gained from an inertial measurement unit (IMU), unit: rad
  - ω: a measurement of the rotational speed from an angular rate (gyro) sensor, unit: rad/s
- Define robot state space: S
  - $S = \{s\}$, $s = (x, y, \theta, \omega)$
  - (x, y): the coordinate of the centerpoint of the wheels, $x \in [0, L]$, $y \in [0, W]$, unit: mm
  - θ: the angle between $+X$ and the orientation of the robot, $\theta \in [0, 2\pi)$, unit: rad
  - ω: the angular velocity of the robot, unit: rad/s, dependent on inputs (w_1, w_2):

    $$\omega = (w_2 - w_1) * C/width. \tag{1}$$

    Positive ω means the robot turns in the direction of anti-clockwise, negative ω means clockwise.

- System dynamics
  When time evolves with time step dt, the state updates from s_t to the new state s_{t+1}:

$$s_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \\ \omega_t \end{pmatrix}, \; s_{t+1} = \begin{pmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \\ \omega_{t+1} \end{pmatrix} = \begin{pmatrix} x_t + ds\cos(\theta_t + d\theta/2) \\ y_t + ds\sin(\theta_t + d\theta/2) \\ \theta_t + d\theta \\ (\omega_2 - \omega_1)\, C/width \end{pmatrix}, \tag{2}$$

  where $ds = (\omega_1 + \omega_2) * C * dt/2$, $d\theta = \omega_{t+1} * dt = (\omega_2 - \omega_1) * C * dt/width$.
  The reason why the difference angle $d\theta$ in the expression of x_{t+1}, y_{t+1} should be divided by 2 is illustrated in Figure 1.

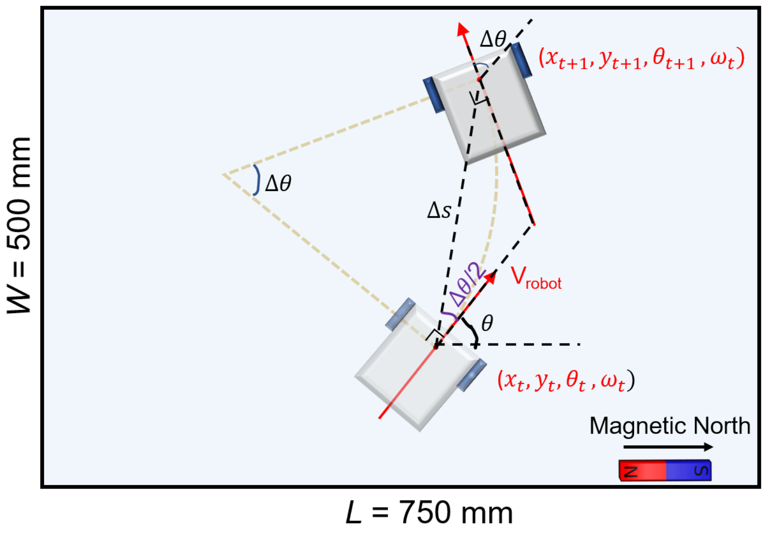

Figure 1: Schematics of robot motion showing the angle variation

  Due to the symmetric geometry feature of a circle, when the orientation of the robot changes a difference angle of $d\theta$ from the original orientation θ, the continuous trajectory changes an angle of $d\theta/2$.

2(b). Include realistic noise terms into the model as appropriate, and numerically quantify their parameters.

- Define process noise: (w_{1e}, w_{2e})
  There may be slippage between the wheels and the floor or sticking in the gears of the motor, so assume the error of the angular velocity of each wheel w_{1e}, w_{2e} are independent zero-mean Gaussians:

$$w_{1e}, w_{2e} \sim \mathcal{N}(0, w_{cov}), \tag{3}$$

  where w_{cov} is the Gaussian covariance of w_{1e} and w_{2e}, dependent only on the rotation servo, in our system the part number is FS90R. According to the question sheet, the standard deviation of the effective angular velocity is 5% of the max motor speed w_{max}, so $w_{cov} = (0.05 w_{max})^2$.

- Define measurement noise: $(d_{1e}, d_{2e}, \theta_e, \omega_e)$
  $(d_{1e}, d_{2e}, \theta_e, \omega_e)$ are sensor errors for each sensor output components $(d_1, d_2, \theta, \omega)$.
  - d_{1e}, d_{2e}: error of distances (d_1, d_2) measured by laser range sensor, unit: mm

$$d_{1e} \sim \mathcal{N}(0, d_{cov} * d_1^2), \tag{4}$$
$$d_{2e} \sim \mathcal{N}(0, d_{cov} * d_2^2), \tag{5}$$

    where d_{cov} is the relative variance of d_e, decided by the property of the laser range sensor VL53L0X in our system. Given that the relative standard deviation is 6% with regard to the measured distance [4], so $d_{cov} = 0.06^2$, $d_{ie} \sim \mathcal{N}(0, (0.06 d_i)^2)$ (for $i = 1, 2$).
  - θ_e: error of angle θ measured by gyroscope, unit: rad

$$\theta_e \sim \mathcal{N}(0, \theta_{cov}), \tag{6}$$

    where θ_{cov} is the variance of the angle error θ_e, decided by the standard deviation of the gyroscope on MPU-9250, which is 0.1° [5], so $\theta_{cov} = (0.1°/180° * \pi)^2$.
  - ω_e: error of angular velocity of the robot, unit: rad/s

$$\omega_e \sim \mathcal{N}(\omega_{bias}, \omega_{cov}), \tag{7}$$

    where ω_{bias}, ω_{cov} are the bias and the variance of the angular velocity error ω_e. According to the product specification of the gyroscope on MPU-9250 [5], $\omega_{bias} = 0$ (unit: rad/s) for now, and $\omega_{cov} = (0.1°/180° * \pi)^2$.

2(c). Create a Kalman Filter (KF) based state estimator to take the motor commands and sensor measurements and generate a state estimate. You will likely want to to implement an Extended Kalman Filter (EKF), but you could choose an Unscented Kalman Filter (UKF) instead. Be sure to explain which algorithm you chose and why, and generate the resulting mathematical expressions.

We applied Extended Kalman Filter (EKF) for the following reasons:

(1) The system is nonlinear, so we could choose from EKF or UKF instead of simple KF.

(2) UKF linearizes a nonlinear function through a linear regression between n points drawn from the prior distribution, while EKF only linearizes the estimation around the current state using the partial derivatives of the process and measurement functions, so UKF is more computationally heavy than EKF.

(3) The Jacobians in EKF could be accurately, numerically calculated, making it convenient to obtain the operational matrices in the algorithm.

To implement Extended Kalman Filter (EKF), we first write down the non-linear stochastic difference equations for system model:

$$s_{t+1} = f(s_t, u_t, w_t), \tag{8}$$

and the measurement function:

$$y_t = h(s_t, v_t), \tag{9}$$

where the state vector $s_t,\ s_{t+1} \in \mathbb{R}^4$, the input vector $u_t \in \mathbb{R}^2$, and the random variable $w_t \in \mathbb{R}^2$ and $v_t \in \mathbb{R}^5$ represent the process and measurement noise vector. The non-linear function f relates the state s from time t to the next time step $t+1$. The non-linear function h relates the state s_t to the measurement y_t.

Algorithm 1: The non-linear stochastic difference function f

```
# Update state from t to t + dt
def stateTimeEvolution(s, inputs, w_cov = 0):
    # Given: current state s = (x, y, theta, omega)
    # Given: inputs from actuators, inputs = (w1, w2), angular velocities of
        the two wheels
    # Return: updated state s' = (x', y', theta', omega') after evolving for
        time dt

    w1, w2 = inputs
    # w1_e, w2_e: error of effective angular velocity, ~ N(0, w_cov)
    w1_e = np.random.normal(0, math.sqrt(w_cov))
    w2_e = np.random.normal(0, math.sqrt(w_cov))

    # v1, v2: velocity of the left and right wheel, unit: mm/s
    v1 = (w1 + w1_e) * C
    v2 = (w2 + w2_e) * C

    # Effective linear and angular velocity of the robot (angular velocity
        direction: positive: anti-clockwise, negative: clockwise)
    v_robot = (v1 + v2) / 2          # unit: mm/s
    w_robot = (v2 - v1) / width      # unit: rad/s

    # Linear and angular displacements in time dt
    ds = v_robot * dt
    d_theta = w_robot * dt

    theta = s[2]
    dx = ds * math.cos(theta + d_theta / 2)
    dy = ds * math.sin(theta + d_theta / 2)

    # When t -> t + dt: x -> x + dx, y -> y + dy, theta -> (theta + d_theta) %
         2pi
    s_new = [s[0] + dx, s[1] + dy, (s[2] + d_theta) % (2 * math.pi), w_robot]
    return s_new
```

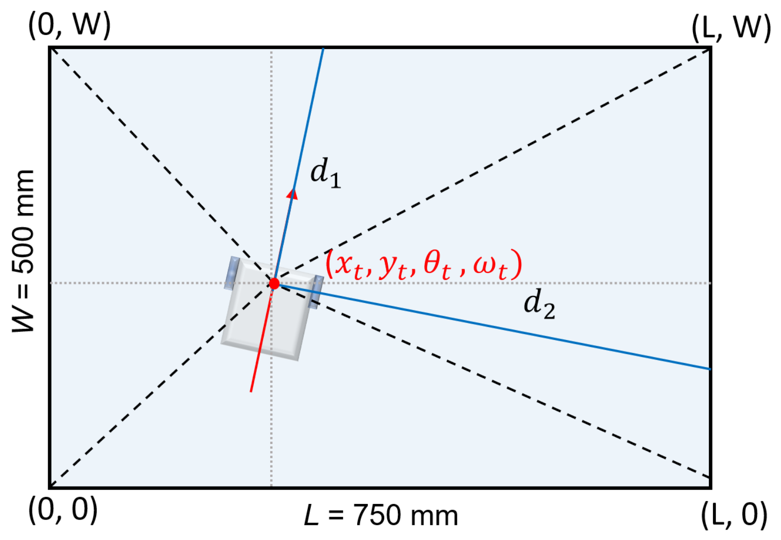

Figure 2: Schematics of robot motion showing the angle variation in Algorithm 2.

Figure 2 demonstrates how to calculate output distances d_1, d_2 from the robot state x, y, θ at any time t in four steps:

Step 1: Calculate the angles between magnetic north (direction of +X) and the line connecting the robot and four corners of the wall.

$$\begin{cases} \theta_1 = \arctan((W-y)/(L-x)) \\ \theta_2 = \arctan((W-y)/(-x)) \\ \theta_3 = \arctan((-y)/(-x)) + 2\pi \\ \theta_4 = \arctan((-y)/(L-x)) + 2\pi \end{cases} \tag{10}$$

Keep it in mind that $0 \leq \theta, \theta_1, \theta_2, \theta_3, \theta_4 < 2\pi$.

Step 2: Divide the intersection point on the wall into four regions (up, left, bottom, right), depending on the angle of the laser with regard to the angles calculated in Step 1. So the equations for d_1, d_2 would be piece-wise functions, having different expressions for $\theta \in [\theta_1, \theta_2),\ [\theta_2, \theta_3),\ [\theta_3, \theta_4),\ [\theta_4, 2\pi) \cup [0, \theta_1)$.

Step 3: Calculate the distance d_1 to the intersection on the wall in front of the robot.

$$d_1 = \begin{cases} \dfrac{W-y}{\sin(\theta)} & \theta_1 \leqslant \theta < \theta_2 \\ \dfrac{x}{\cos(\pi-\theta)} & \theta_2 \leqslant \theta < \theta_3 \\ \dfrac{y}{\sin(\theta-\pi)} & \theta_3 \leqslant \theta < \theta_4 \\ \dfrac{L-x}{\cos(\theta)} & \theta_4 \leqslant \theta < 2\pi \text{ or } 0 \leqslant \theta < \theta_1 \end{cases} \tag{11}$$

Step 4: Repeat Step 3 to calculate the distance d_2 to the right of the robot.

$$\theta' = \theta - \pi/2 \pmod{2\pi}$$

$$d_2 = \begin{cases} \dfrac{W-y}{\sin(\theta')} & \theta_1 \leqslant \theta' < \theta_2 \\ \dfrac{x}{\cos(\pi-\theta')} & \theta_2 \leqslant \theta' < \theta_3 \\ \dfrac{y}{\sin(\theta'-\pi)} & \theta_3 \leqslant \theta' < \theta_4 \\ \dfrac{L-x}{\cos(\theta')} & \theta_4 \leqslant \theta' < 2\pi \text{ or } 0 \leqslant \theta' < \theta_1 \end{cases} \tag{12}$$

With Equation (11, 12), the algorithm for function h is as follows:

Algorithm 2: The non-linear measurement function h

```
def stateToSensor(s, d_cov = 0, theta_cov = 0, omega_bias = 0, omega_cov = 0):
    # Given: robot state s = (x, y, theta, omega)
    # Return: sensor outputs (d1_s, d2_s, theta_s, omega_s)

    x, y, theta, omega = s[0], s[1], s[2], s[3]

    # theta_e: error of measured robot orientation
    # theta_s: measured sensor value of robot orientation
    theta_e = np.random.normal(0, math.sqrt(theta_cov))
    theta_s = (theta + theta_e) % (2 * math.pi)

    # Calculate the angles from the robot position to four wall corners
    theta1 = math.atan2(W - y, L - x)                # top-right (L, W)
    theta2 = math.atan2(W - y, -x)                   # top-left (0, W)
    theta3 = math.atan2(-y, -x) + 2 * math.pi        # bottom_left (0, 0)
    theta4 = math.atan2(-y, L - x) + 2 * math.pi     # bottom_right (L, 0)

    # Calculate the distance to the nearest wall in front of the robot
    if  theta1 <= theta < theta2:                    # intersection on the top wall
        d1 = (W - y) / math.sin(theta)
    elif theta2 <= theta < theta3:                   # intersection on the left wall
        d1 = x / math.cos(math.pi - theta)
    elif theta3 <= theta < theta4:                   # intersection on the bottom wall
        d1 = y / math.sin(theta - math.pi)
    else:                                            # intersection on the right wall
        d1 = (L - x) / math.cos(theta)

    # Calculate the distance to the nearest wall to the right of the robot
    theta = (theta - math.pi / 2) % (2 * math.pi)
    if  theta1 <= theta < theta2:                    # intersection on the top wall
        d2 = (W - y) / math.sin(theta)
    elif theta2 <= theta < theta3:                   # intersection on the left wall
        d2 = x / math.cos(math.pi - theta)
    elif theta3 <= theta < theta4:          # intersection on the bottom wall
        d2 = y / math.sin(theta - math.pi)
    else:                                   # intersection on the right wall
        d2 = (L - x) / math.cos(theta)

    # Calculate noises
    d1_e = np.random.normal(0, math.sqrt(d_cov)) * d1
    d2_e = np.random.normal(0, math.sqrt(d_cov)) * d2
    omega_e = np.random.normal(omega_bias, math.sqrt(omega_cov))
    d1_s = d1 + d1_e
    d2_s = d2 + d2_e
    omega_s = omega + omega_e
    return [d1_s, d2_s, theta_s, omega_s]
```

To linearize an estimation about Equation (8) and Equation (9), we have linear approximation equations:

$$\widetilde{s}_{t+1} \approx F_t \widetilde{s}_t + W_t w_t, \tag{13}$$

$$\widetilde{y}_t \approx H_t \widetilde{s}_t + V_t v_t, \tag{14}$$

where

- $\widetilde{s}_{t+1}$ and $\widetilde{s}_t$ are the linear approximate state vector at time t and $t+1$:

$$\widetilde{s}_t = \begin{pmatrix} x_t \\ y_t \\ \theta_t \\ w_t \end{pmatrix}, \widetilde{s}_{t+1} = \begin{pmatrix} x_{t+1} \\ y_{t+1} \\ \theta_{t+1} \\ w_{t+1} \end{pmatrix} \approx \begin{pmatrix} x_t + k_1 z_1 \cos(\theta_t + k_2 z_2) \\ y_t + k_1 z_1 \sin(\theta_t + k_2 z_2) \\ \theta_t + 2k_2 z_2 \\ k_2 C/width \end{pmatrix}, \tag{15}$$

  where $z_1 = Cdt/2$, $z_2 = Cdt/(2 * width)$, $k_1 = (w_{1t} + w_{1et}) + (w_{2t} + w_{2et})$, $k_2 = (w_{2t} + w_{2et}) - (w_{1t} + w_{1et})$, derived from Equation (2, 3).
- $u_t = (w_{1t}, w_{2t})$ is the inputs at time t.
- $w_t = (w_{1et}, w_{2et})$ is the process noise as defined in Equation (3).
- F_t is the Jacobian matrix of partial derivatives of f with respect to state s_t:

$$F_t[i,j] = \frac{\partial f_i(s_t, u_t, 0)}{\partial s_j} = \frac{\partial \widetilde{s}_{t+1}[i]}{\partial s_t[j]}. \tag{16}$$

  In our system, according to Equation (15), F_t can be numerically calculated as:

$$F_t = \begin{pmatrix} 1 & 0 & -ds * \sin(\theta + d\theta/2) & 0 \\ 0 & 1 & ds * \cos(\theta + d\theta/2) & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}, \tag{17}$$

  where $ds = (\omega_1 + \omega_2) * C * dt/2$, $d\theta = (\omega_2 - \omega_1) * C * dt/width$.

- W_t is the Jacobian matrix of partial derivatives of f with respect to process noise w_t:

$$W_t[i,j] = \frac{\partial f_i(s_t, u_t, 0)}{\partial w_j} = \frac{\partial \tilde{s}_{t+1}[i]}{\partial w_{et}[j]}. \tag{18}$$

In our system, according to Equation (15), W_t can be numerically expressed as:

$$W_t = \begin{pmatrix} \frac{\partial \tilde{s}_{t+1}}{\partial w_{1et}} & \frac{\partial \tilde{s}_{t+1}}{\partial w_{2et}} \end{pmatrix} = \begin{pmatrix} z_1 \cos\alpha + z_2\, ds \sin\alpha & z_1 \cos\alpha - z_2\, ds \sin\alpha \\ z_1 \sin\alpha - z_2\, ds \cos\alpha & z_1 \sin\alpha + z_2\, ds \cos\alpha \\ -2z_2 & 2z_2 \\ -C/width & C/width \end{pmatrix}, \tag{19}$$

where $\alpha = \theta + d\theta/2$, $z_1 = C * dt/2$, $z_2 = C * dt/(2 * width)$, $ds = (w_1 + w_2) * C * dt/2$, $d\theta = (w_2 - w_1) * C * dt/width$.

- To facilitate future calculation, we introduce a covariance matrix Q_t for noise w_{et} as such:

$$Q_t = \mathbb{E}\left[w_{et} w_{et}^{\mathbf{T}}\right], \tag{20}$$

and in our system numerically,

$$Q_t = \begin{pmatrix} w_{cov} & 0 \\ 0 & w_{cov} \end{pmatrix}. \tag{21}$$

- In Equation (14), $\tilde{y}_t$ is the linear approximate measurement output vector at time t.

$$\tilde{y}_t = \begin{pmatrix} d_{1t} \\ d_{2t} \\ \theta_t \\ \omega_t \end{pmatrix}, \tag{22}$$

where d_{1t}, d_{2t} can be found in Equation (11, 12), the measured θ_t and ω_t are not directly dependent on the current state s_t.

- H_t is the Jacobian matrix of partial derivatives of h with respect to state s_t:

$$H_t[i,j] = \frac{\partial h_i(s_t, 0)}{\partial s_j} = \frac{\partial \tilde{y}_t[i]}{\partial s_t[j]}, \tag{23}$$

so typically, we can use the abstract form of Equation (26) to represent H_t in our system for state s at time t:

$$H_t = \begin{pmatrix} h_{00} & h_{01} & h_{02} & 0 \\ h_{10} & h_{11} & h_{12} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{24}$$

For convenience, we can also denote the sub-matrix of interest as:

$$H' = \begin{pmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \end{pmatrix} = \begin{pmatrix} \mathbf{h}_0 \\ \mathbf{h}_1 \end{pmatrix}, \tag{25}$$

where the vector $\mathbf{h}_0 = \left(\frac{\partial d_1}{\partial x}, \frac{\partial d_1}{\partial y}, \frac{\partial d_1}{\partial \theta}\right)$, $\mathbf{h}_1 = \left(\frac{\partial d_2}{\partial x}, \frac{\partial d_2}{\partial y}, \frac{\partial d_2}{\partial \theta}\right)$.

Since d_1, d_2 are piece-wise functions divided into four regions by the orientation θ (see Figure 2), the explicit expression of $\mathbf{h}_0$, $\mathbf{h}_1$ will also be divided into four regions, and could be derived from Equation (10-12) as follows:

- For wall in the front of the robot: $\alpha = \theta$ $\quad d = d_1 \quad \mathbf{h} = \mathbf{h}_0$,
  For wall to the right of the robot: $\alpha = \theta - \pi/2 \pmod{2\pi}$ $\quad d = d_2 \quad \mathbf{h} = \mathbf{h}_1$.
- The intersection point is on the top wall ($\theta_1 \leqslant \alpha < \theta_2$), $d = (W-y)/\sin\alpha$,

$$\mathbf{h} = \left(0, -\frac{1}{\sin\alpha}, -\frac{(W-y)\cos\alpha}{\sin^2\alpha}\right). \tag{26a}$$

- The intersection point is on the left wall ($\theta_2 \leqslant \alpha < \theta_3$), $d = x/\cos(\pi-\alpha)$,

$$\mathbf{h} = \left(-\frac{1}{\cos\alpha}, 0, -\frac{x\sin\alpha}{\cos^2\alpha}\right). \tag{26b}$$

- The intersection point is on the bottom wall ($\theta_3 \leqslant \alpha < \theta_4$), $d = y/\sin(\alpha-\pi)$,

$$\mathbf{h} = \left(0, -\frac{1}{\sin\alpha}, \frac{y\cos\alpha}{\sin^2\alpha}\right). \tag{26c}$$

- The intersection point is on the right wall ($\theta_4 \leqslant \alpha < 2\pi$ or $0 \leqslant \alpha < \theta_1$), $d = (L-x)/\cos\alpha$,

$$\mathbf{h} = \left(-\frac{1}{\cos\alpha}, 0, \frac{(L-x)\sin\alpha}{\cos^2\alpha}\right). \tag{26d}$$

- V_t is the Jacobian matrix of partial derivatives of h with respect to noise v_t:

$$V_t[i,j] = \frac{\partial h_i(s_k, 0)}{\partial v_j} = \frac{\partial \widetilde{y}_t[i]}{\partial v_t[j]}. \tag{27}$$

  Since the V_t term is trivial in our system's Kalman Filter updates, we will neglect it.

- To facilitate future calculation, we introduce a covariance matrix R_t for noise v_t as such:

$$R_t = \mathbb{E}\left[v_t v_t^{\mathbf{T}}\right], \tag{28}$$

  and in our system,

$$R_t = \begin{pmatrix} d_{cov} * d_1^2 & 0 & 0 & 0 \\ 0 & d_{cov} * d_2^2 & 0 & 0 \\ 0 & 0 & \theta_{cov} & 0 \\ 0 & 0 & 0 & \omega_{cov} \end{pmatrix}. \tag{29}$$

- Now we can implement EKF time update:

$$\begin{aligned} s_{t+1}^- &= f(s_t, u_t, 0) \\ P_{t+1}^- &= F_t P_t F_t^{\mathbf{T}} + W_t Q_t W_t^{\mathbf{T}} \end{aligned} \tag{30}$$

  The time update equations in Equation (30) project the state s and covariance estimates P from the previous time step t to the next time step $t+1$ ($s_t \to s_{t+1}^-$, $P_t \to P_{t+1}^-$). Again function f comes from Equation (8), F_t and W_t are the process Jacobians from Equation (17, 19), and Q_t is the process noise covariance from Equation (21).

- And EKF measurement update:

$$\begin{aligned} K_t &= H_t P_{t+1}^- H_t^{\mathbf{T}} + R_t \\ s_{t+1} &= s_{t+1}^- + P_{t+1}^- H_t^{\mathbf{T}} K_t^{-1}(y_t - h(s_{t+1}^-, 0)) \\ P_{t+1} &= P_{t+1}^- - P_{t+1}^- H_t^{\mathbf{T}} K_t^{-1} H_t P_{t+1}^- \end{aligned} \tag{31}$$

  The measurement update equations in Equation (31) correct the state and covariance estimates with the measurement y_t ($s_{t+1}^- \to s_{t+1}$, $P_{t+1}^- \to P_{t+1}$). Again function h comes from Equation (9), H_t is the measurement Jacobian at time step t from Equation (26), and R_t is the measurement noise covariance from Equation (29). Note that we already neglect the trivial term of the Jacobian matrix V_t in the expression of K_t above.

- A complete picture of the operation of the Extended Kalman Filter (EKF) is shown in Figure 3, and the algorithm is shown in Algorithm 3.

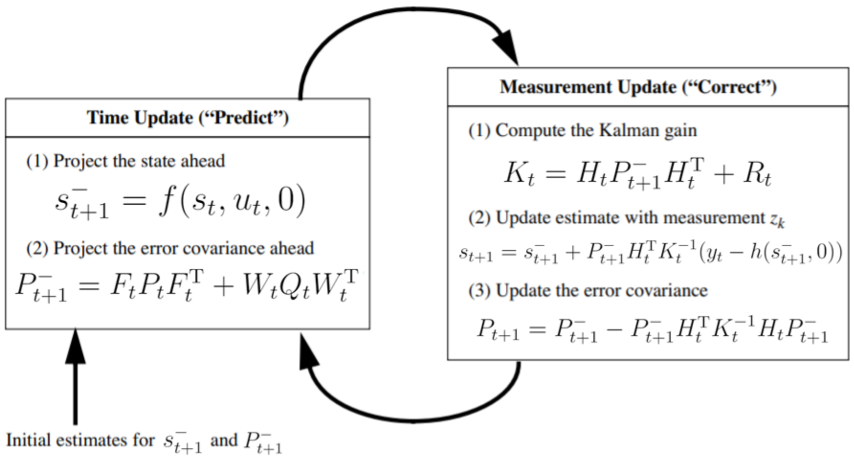
Time Update ("Predict")
(1) Project the state ahead
$s_{t+1}^- = f(s_t, u_t, 0)$
(2) Project the error covariance ahead
$P_{t+1}^- = F_t P_t F_t^{\mathrm{T}} + W_t Q_t W_t^{\mathrm{T}}$
Measurement Update ("Correct")
(1) Compute the Kalman gain
$K_t = H_t P_{t+1}^- H_t^{\mathrm{T}} + R_t$
(2) Update estimate with measurement z_k
$s_{t+1} = s_{t+1}^- + P_{t+1}^- H_t^{\mathrm{T}} K_t^{-1}(y_t - h(s_{t+1}^-, 0))$
(3) Update the error covariance
$P_{t+1} = P_{t+1}^- - P_{t+1}^- H_t^{\mathrm{T}} K_t^{-1} H_t P_{t+1}^-$
Initial estimates for s_{t+1}^- and P_{t+1}^-

Figure 3: A complete pipeline of Extended Kalman Filter [1].

Algorithm 3: Extended Kalman Filter

```
def EKF(s0, P0, inputs, w_cov, d_cov, theta_cov, omega_bias, omega_cov,
    Fmatrix, Wmatrix, Qmatrix, Hmatrix, Rmatrix, timeUpdate =
    stateTimeEvolution, measureUpdate = stateToSensor):
      # Given:
          # s0: initial state (x, y, theta, omega)
          # P0: initial covariance matrix of the state, dim(P0) = dim(s) *
              dim(s) = 4 * 4
          # inputs: a list of control input sequences [(w1, w2)]
          # w_cov, d_cov, theta_cov, omega_bias, omega_cov: noise parameters
              , as defined in Problem 2(b)
          # Fmatrix, Wmatrix, Qmatrix, Hmatrix, Rmatrix: matrices in the
              form of functions, as defined along with process and
              measurement model
          # timeUpdate: process model, s' = f(s, u, w), return state (x, y,
              theta, omega)
          # measureUpdate: measurement model, y = h(s, v), return outputs (
              d1, d2, theta, omega)
     # Return:
          # traj: real trajectory of the robot {(x, y, theta, omega)},
              generated from inputs and time update function, with process
              noise
          # obs: observed traces {(d1, d2, theta, omega)}, gained from
              measurement update function, with measurement noise
          # exp: expected trajectory {(x', y', theta', omega')}, gained from
               EKF algorithms, to estimate the real trajectory

     # Number of robot motions, same as the length of the input sequences
     step = len(inputs)

     # Initialization
     traj, exp = [s0], [s0]
     obs =[]
     s = s0
     P = P0

     # Trajectory evolution with time step of dt
     for i in range(step):
          s = traj[i]
          w = inputs[i]
          F = Fmatrix(s, w)
          W = Wmatrix(s, w)
          Q = np.dot(W, Qmatrix(w_cov)).dot(W.T)    # Q(t+1)=W(t)Q(t)W(t)^T

          # Real state
          s1 = timeUpdate(s, w, w_cov)    # s(t+1)=f(s(t),inputs(t),
              input_noise(t))
          # Get observation for s1
          ob = measureUpdate(s1, d_cov, theta_cov, omega_bias, omega_cov) #
              y(t)=h(s(t),output_noise(t))
          obs.append(ob)
```

```
        traj.append(s1)

        # Time update estimate
        s = exp[i]
        s1 = timeUpdate(s, w, 0)          # s(t+1)=f(s(t),inputs(t),0)
            without noise
        P = np.dot(F, P).dot(F.T) + Q    # P(t+1)=F(t)P(t)F(t)^T+W(t)Q(t)W(
            t)^T
        # exp.append(s1)

        # Observation update estimate
        R = Rmatrix(s1)
        H = Hmatrix(s1)
        K = np.dot(H, P).dot(H.T) + R    # K(t)=H(t)P(t)H(t)^T+R
        s2 = list(np.array(s1) + np.dot(P, H.T).dot(np.linalg.inv(K)).dot(
            np.array(ob) - np.array(measureUpdate(s1,0,0,omega_bias,0))))
               # s(t+1)'=s(t+1)+P(t+1)H(t)^T K^-1 (y(t)-h(s(t),0))
        P = P - np.dot(P, H.T).dot(np.linalg.inv(K)).dot(H).dot(P)
                    # P(t+1)'=P(t+1)-P(t+1)H(t)^T K^-1 H(t) P(t+1)
        exp.append(s2)
    return traj, exp, obs
```

3 Evaluation

3(a). Define and describe several reference trajectories (that in turn generate control input sequences) that capture the abilities and limitations of a state estimator in this environment.

we have created several reference trajectories. For any trajectory we use initial state s_0 and control input sequences *inputs* to generate the trajectory.

- Trajectory 1: Straight forward line in the direction of $+x$

```
s0 = [100, 100, 0, 0]
inputs = [[1, 1] * 40]
```

- Trajectory 2: Straight forward line in the direction of $+y$

```
s0 = [100, 100, pi/2, 0]
inputs = [[1, 1] * 40]
```

- Trajectory 3: Clockwise circular trajectory

```
s0 = [200, 200, 0, 0]
inputs = [[1, 0] * 40]
```

- Trajectory 4: Counter clockwise circular trajectory

```
s0 = [200, 200, 0, 0]
inputs = [[0, 1] * 40]
```

- Trajectory 5: Straight line in the direction of -x

```
s0 = [400, 400, pi, 0]
inputs = [[1, 1] * 50]
```

- Trajectory 6: A straight line in the direction of -y

```
s0 = [400, 400, pi*3/2, 0]
inputs = [[1, 1] * 40]
```

- Trajectory 7: A straight line in the direction of 45 degree from +x

```
s0 = [100, 100, pi/4, 0]
inputs = [[1, 1] * 40]
```

- Trajectory 8: Zigzag trajectory

```
s0 = [300, 300, 0, 0]
inputs = []
for i in range(10):
    inputs.append([1, -1])
for i in range(10):
    inputs.append([1, 1])
for i in range(10):
    inputs.append([-1, 1])
for i in range(10):
    inputs.append([1, 1])
```

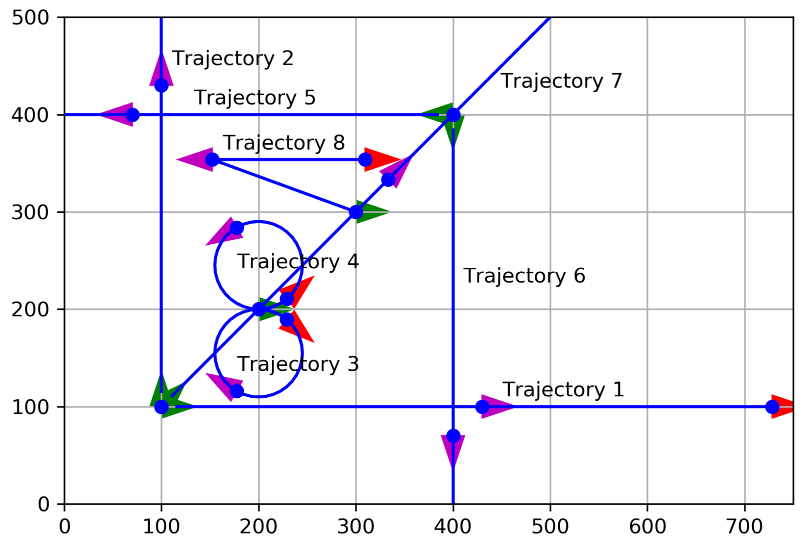

Figure 4: Reference trajectories.

In Figure 4, the green, magenta, and red arrow denotes the beginning, middle, and the end point of every trajectory, respectively. And the direction of every arrow is the same as the orientation of the robot at that state.

3(b). Implement a simulation, including models of your sensor and actuator response (especially including noise), to generate realistic sensor traces given the above control inputs. Present and explain the simulated sensor traces.

We choose an arbitrary trajectory and apply time update function f (described in Algorithm 1 and Equation (15)) to get the real states, then we use get observation function h to obtain the observations.

For example, if s_0 = [100,100,0,0] and inputs = [1, 1] * 5, we obtain the following sensor traces containing sequential states:

[[678.34, 97.14, 6.27, -0.088], [622.86, 98.94, 6.27, 0.012], [666.30, 105.79, 0.0046, 0.11], [645.31, 88.22, 0.025, 0.23], [568.85, 95.16, 0.034, 0.08]]

3(c). Implement your KF based state estimator on these examples, demonstrating the performance (in terms of accuracy and efficiency) of your computed state estimate over time as the robot is issued commanded input sequences. Consider both perfect knowledge of the initial state as well as no knowledge of the initial state on the same traces. Clearly describe the experiments that were run, the data that was gathered, and the process by which you use that data to characterize the performance of your state estimator. Include figures; you may also refer to animations uploaded to your git repo.

We have run all the trajectories in the 3(a) considering both perfect knowledge of the initial state and no knowledge of the initial state. For perfect knowledge of the initial state, the initial covariance matrix $\mathbf{P}_0$ is all zero matrix:

$$P_0 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{32}$$

For no knowledge of the initial state, we consider the robot is uniform distributed in all possible states, so the initial covariance matrix $\mathbf{P}_0$ is:

$$P_0 = \begin{pmatrix} 750/12 & 0 & 0 & 0 \\ 0 & 500/12 & 0 & 0 \\ 0 & 0 & pi * 2/3 & 0 \\ 0 & 0 & 0 & C/width/6 \end{pmatrix} \tag{33}$$

Figure 5 to Figure 20 display the trajectories. In the figures, red trajectories are estimated trajectories and blue trajectories are real trajectories. To show the errors, the radius of green circles is 3σ of the state according to the estimated covariance matrix.

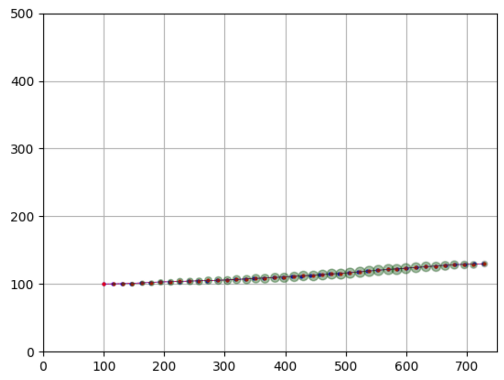

Figure 5: Perfect knowledge of s_0

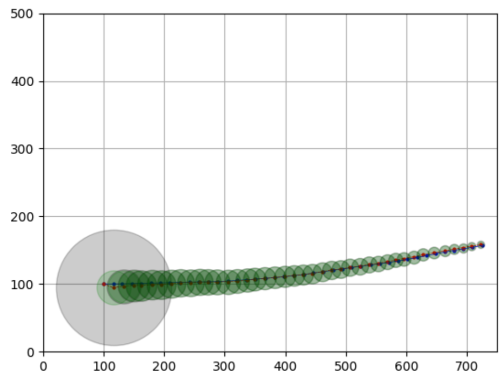

Figure 6: No knowledge of s_0

Trajectory No.1: Straight forward line in the direction of $+x$.

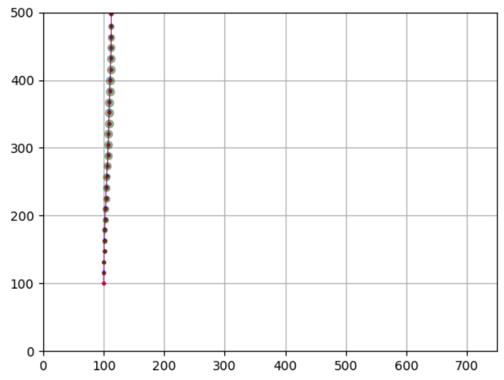

Figure 7: Perfect knowledge of s_0

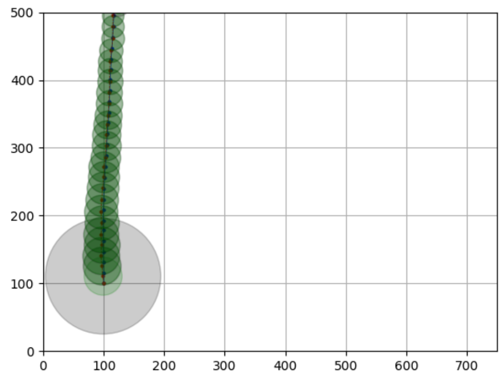

Figure 8: No knowledge of s_0

Trajectory No.2: Straight line trajectory heading in the direction of $+y$.

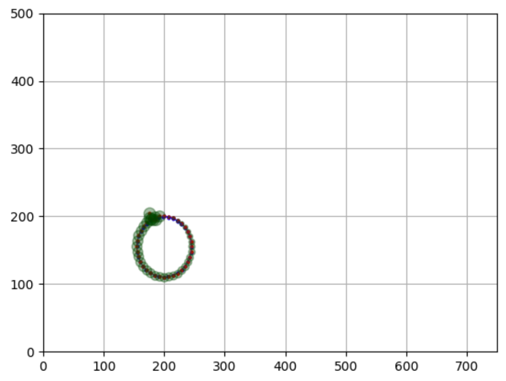

Figure 9: Perfect knowledge of s_0

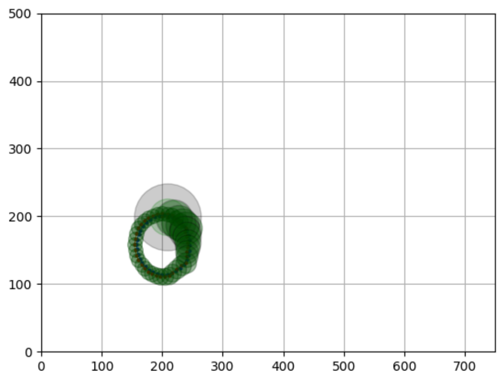

Figure 10: No knowledge of s_0

Trajectory No.3: Clockwise circular trajectory.

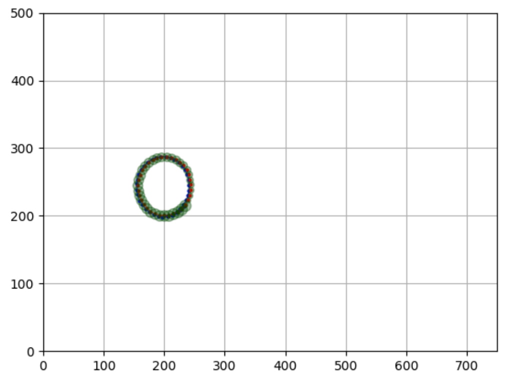

Figure 11: Perfect knowledge of s_0

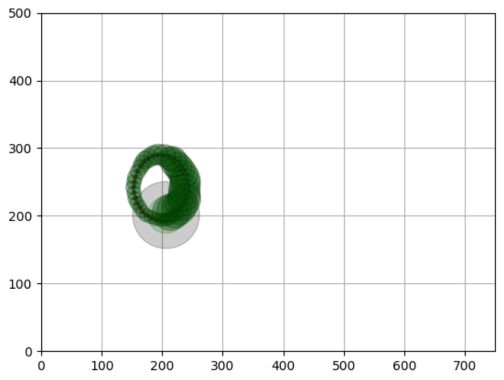

Figure 12: No knowledge of s_0

Trajectory No.4: Counter-clockwise circular trajectory.

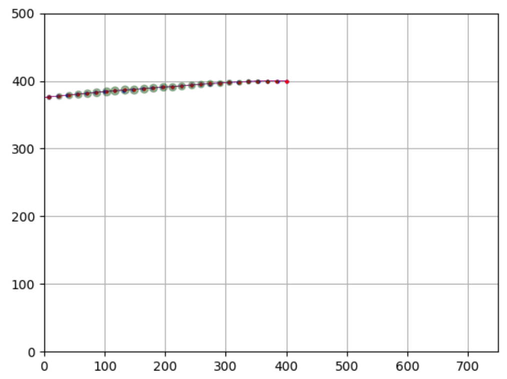
Figure 13: Perfect knowledge of s_0

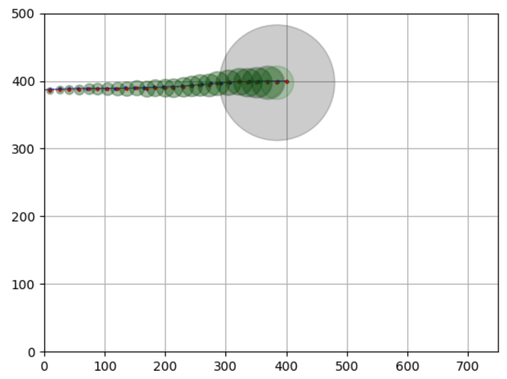
Figure 14: No knowledge of s_0

Trajectory No.5: Straight line heading in $-x$.

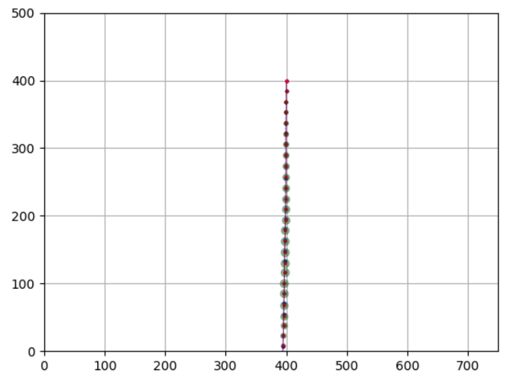
Figure 15: Perfect knowledge of s_0

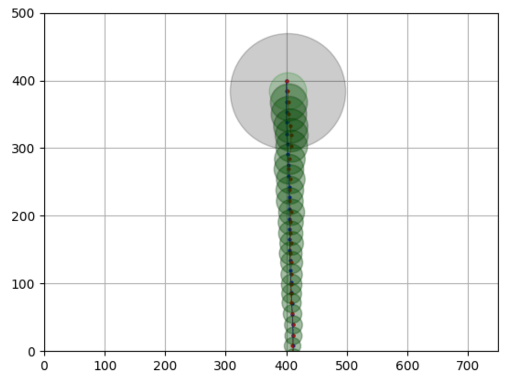
Figure 16: No knowledge of s_0

Trajectory No.6: Straight line heading in $-y$.

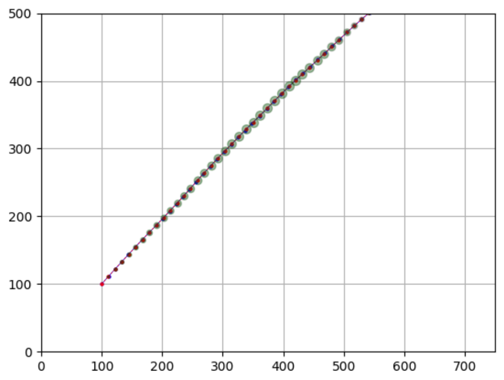
Figure 17: Perfect knowledge of s_0

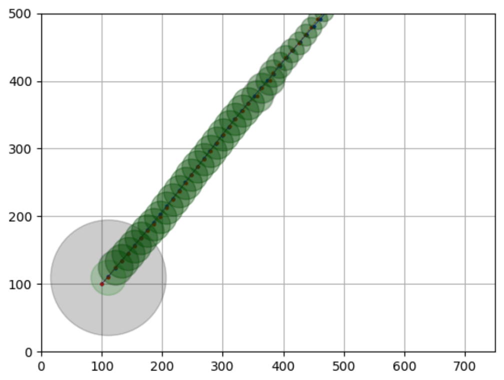
Figure 18: No knowledge of s_0

Trajectory No.7: Straight line heading in the direction 45^o from $+x$.

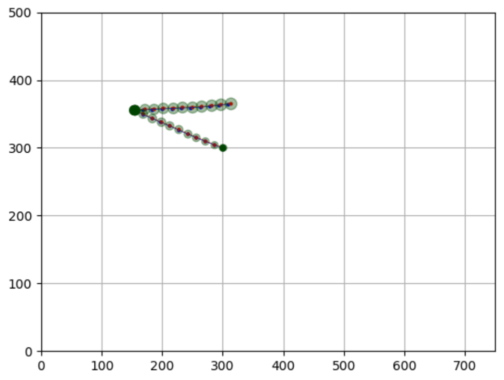
Figure 19: Perfect knowledge of s_0

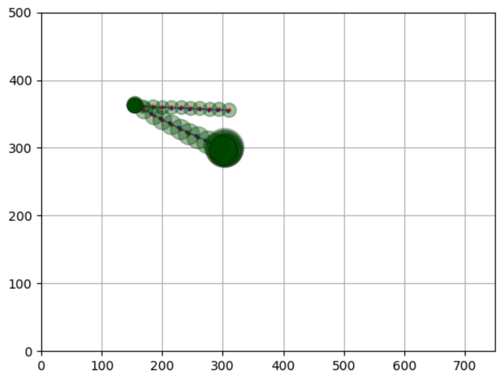
Figure 20: No knowledge of s_0

Trajectory No.8: A complex trajectory.

From the figures, we can conclude that if we have perfect knowledge of the initial state, the covariance is small at first and then become large before converging to small values in the end. Whereas when we have no knowledge of the initial state, the covariance is large at first and then converge to a smaller value towards the end.

To characterize the performance of state estimator, we use mean error and std error between both real trajectory and estimated trajectory. The results are shown below:

- trajectory1 with all knowledge: (1.60, 1.89)
- trajectory1 with no knowledge:(2.5, 3.04)

- trajectory2 with all knowledge:(1.1, 1.3)
- trajectory2 with no knowledge: (4.48, 5.0)
- trajectory3 with all knowledge: (1.2, 1.3)
- trajectory3 with no knowledge:(3.6, 3.8)
- trajectory4 with all knowledge:(2.2, 2.3)
- trajectory4 with no knowledge:(3.6, 4.27)
- trajectory5 with all knowledge:(0.8, 1.0)
- trajectory5 with no knowledge:(2.7, 3.0)
- trajectory6 with all knowledge:(1.0, 1.3)
- trajectory6 with no knowledge:(5.5, 8.3)
- trajectory7 with all knowledge:(1.4, 1.5)
- trajectory7 with no knowledge:(2.6, 3.6)
- trajectory8 with all knowledge:(1.6, 2.1)
- trajectory8 with no knowledge:(2.4, 2.8).

3(d). How might you accommodate more realistic system models? For example, the gyro sensor does not have zero-mean additive noise; rather, the mean of the gyro noise is a slowly varying non-zero bias value. The actuator uncertainty (process noise) typically depends on the commanded speed itself. Describe the updates to the system model in these and other cases, and how that would impact your state estimator.

We consider the gyro sensor has a additive bias noise that varies linearly with respect to the time, that is $\omega_{bias} = 0.001 * t$, also we consider the actuator uncertainty depending on the commanded speed itself, that is $w_e \sim \mathcal{N}(0, w_{cov} * w * w)$.

Figure 20 and 21 show the trajectories of new system model. The mean error and std error between real trajectory and estimated trajectory of the new system are (2.06, 2.65) and (4.48, 4.85) for all knowledge and no knowledge which are larger than old one (1.35, 1.54) and (2.63, 3.59). This is obvious because now we have more noise in the system.

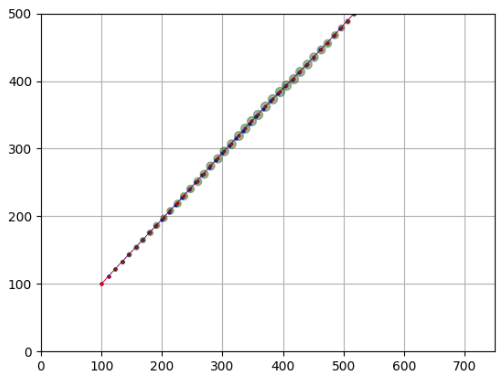

Figure 21: Perfect knowledge of s_0

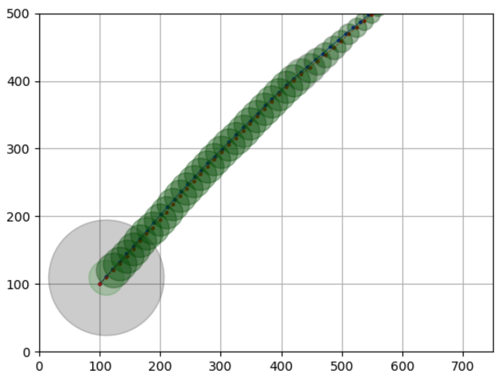

Figure 22: No knowledge of s_0

Updated system model with gyro sensor that has a slowly varying non-zero bias.

3(e). Qualitatively describe some conclusions about the effectiveness of your state estimator for potential tasks your robot may encounter, and trade-offs regarding yours vs other possible state estimation algorithms.

For this lab, we employed the extended Kalman filter that requires linearization as our state estimator, the advantage of it is simple and fast. However, it is not an optimal estimator when the measurement or the state transition model is nonlinear. When the initial state is unknown, the EKF might diverge due to linearization. The estimated covariance matrix of the EKF could also underestimate the actual covariance matrix. Other state estimators such as particle filter will be more suitable in cases where the system model is inaccurate, but such Monte Carlos methods can be more computationally expensive.

Reference

[1] Gary Bishop, and Greg Welch. “An introduction to the kalman filter.” *Proc of SIGGRAPH*, Course 8.27599-23175 (2001): 41.

[2] Alex Becker. Introduction of Kalman Filter. *Kalman Filter Tutorial* (2018). From: https://www.kalmanfilter.net/default.aspx

[3] Terejanu, Gabriel A. “Unscented kalman filter tutorial.” *University at Buffalo, Buffalo* (2011).

[4] VL53L0X Datasheet - production data. *STMicroelectronics* (April 2018). 27-28. From: https://www.st.com/resource/en/datasheet/vl53l0x.pdf

[5] MPU-9250 Product Specification (Revision 1.1). *InvenSense Inc.* (Released: 06/20/2016). 8. From: https://www.invensense.com/wp-content/uploads/2015/02/PS-MPU-9250A-01-v1.1.pdf