Robotic Science and Systems - Exercise 3

Kevin Robb

1 Kalman Filtering

1.a 1-D displacement

A robot is at an unknown position $x \in \mathbb{R}$ on a 1-D line. At each time t, it is commanded to move by $u_t = \Delta x$ and is perturbed by some unknown transition noise $v_t \sim \mathcal{N}(0, \sigma_v^2)$ s.t. $x_{t+1} = x_t + u_t + v_t$. After moving, we get a measurement z_t based on the robot's new position x_{t+1}, which is perturbed by some unknown observation noise $w_{t+1} \sim \mathcal{N}(0, \sigma_w^2)$, s.t. $z_{t+1} = x_{t+1} + w_{t+1}$

With the given parameter values, we can propagate the initial state $x_0 \sim \mathcal{N}(0, 1)$ forward to the posterior distribution on x_3. For this example we have $f = g = h = 1$.

Initially, we have $\hat{x}_0 = 0$, $\hat{\sigma}_0^2 = 1$. Going forward one step yields

$$\hat{x}_1^+ = x_0 + u_0 = 0 - 0.5 = -0.5$$
$$\hat{\sigma}_1^{2+} = \hat{\sigma}_0^2 + \hat{\sigma}_v^2 = 1 + 0.04 = 1.04$$
$$\nu_1 = z_1 - \hat{x}_1^+ = -0.7 + 0.5 = -0.2$$
$$K_1 = \frac{\hat{\sigma}_1^{2+}}{\hat{\sigma}_1^{2+} + \sigma_w^2} = \frac{1.04}{1.04 + 0.01} = 0.9905$$
$$\hat{x}_1 = \hat{x}_1^+ + K_1 \cdot \nu_1 = -0.5 + 0.9905 \cdot -0.2 = -0.6981$$
$$\hat{\sigma}_1^2 = (1 - K_1)\hat{\sigma}_1^{2+} = (1 - 0.9905) \cdot 1.04 = 0.00988$$

We can then take the results of this step and propagate to $t = 2$. This is tedious, so I've written a python script to implement this simple iterative algorithm. After verifying it matches this result for $t = 1$, I ran it for the desired remaining timesteps. The code is shown below; the resulting posterior distribution for the state at timestep $t = 3$ is $x_3 \sim \mathcal{N}(0.939, 0.00829)$. This is more accurate than I would have computed by hand, since the program does not round off each step to a conveniently small number of digits.

```python
#!/usr/bin/env python3

# this code will perform a few iterations of a simple 1D Kalman filter using given parameters.

## Global Variables ##
sig_v = 0.04; sig_w = 0.01
f=1; g=1; h=1
####

def iter_kf(x_0, sig_0, u, z):
    """
    @param x_0, sig_0 the prior (est. at time t)
    @param u the control input at time t
    @param z the measurement at time t+1
    @return x, sig the posterior (est. at time t+1)
    """
    x_predicted = f*x_0 + g*u
    sig_predicted = f**2 * sig_0 + sig_v
    innovation = z - h * x_predicted
    k_gain = sig_predicted * h / (h**2 * sig_predicted + sig_w)
    x = x_predicted + k_gain * innovation
    sig = (1 - k_gain * h) * sig_predicted
    return x, sig

# index corresponds to timestep for u and z.
# control inputs.
u = [-0.5, 1.2, 0.3]
# measurements.
z = [None, -0.7, 0.6, 0.95]
# current state at t=0.
x = 0; sig = 1
# iterate to get the state at timestep 3.
for i in range(3):
    x, sig = iter_kf(x, sig, u[i], z[i+1])
print(x, sig)
```

1.b 2-D displacement

This problem has very similar structure as the previous, with matrices and column vectors replacing our scalars and variables. The relevant set of equations becomes

$$\hat{x}_1^+ = \hat{x}_0 + u_0$$
$$\hat{P}_1^+ = \hat{P}_0 + V$$
$$\nu_1 = z_1 - \hat{x}_1^+$$
$$K_1 = \hat{P}_1^+(\hat{P}_1^+ + W)^{-1}$$
$$\hat{x}_1 = \hat{x}_1^+ + K_1 \cdot \nu_1$$
$$\hat{P}_1 = (I_2 - K_1)\hat{P}_1^+$$

Here I_2 is the 2×2 identity matrix. This is even more tedious than the 1D case, so I've modified my python script and included the code below.

```
#!/usr/bin/env python3

# this code will perform a few iterations of a simple 2D Kalman filter using given parameters.
import numpy as np
from numpy.linalg import inv

## Global Variables ##
V = np.array([[0.04, 0],[0, 0.09]])
W = np.array([[0.01, 0],[0, 0.02]])
F=np.eye(2); G=np.eye(2); H=np.eye(2)
####

def iter_kf(x_0, P_0, u, z):
    """
    @param x_0, P_0 the prior (est. at time t)
    @param u the control input at time t
    @param z the measurement at time t+1
    @return x, P the posterior (est. at time t+1)
    """
    x_predicted = F @ x_0 + G @ u
    P_predicted = F @ P_0 @ F.T + V
    innovation = z - H @ x_predicted
    K_gain = P_predicted @ H.T @ inv(H @ P_predicted @ H.T + W)
    x = x_predicted + K_gain @ innovation
    P = (np.eye(2) - K_gain @ H) @ P_predicted
    return x, P

# index corresponds to timestep for u and z.
# control inputs.
u = [np.array([[-0.5,0.3]]).T,
    np.array([[1.2,-0.6]]).T,
    np.array([[0.3,0.3]]).T]
# measurements.
z = [None,
    np.array([[-0.7,0.3]]).T,
    np.array([[0.6,0.0]]).T,
    np.array([[0.95,0.15]]).T]
# current state at t=0.
x = np.array([[0,0]]).T
P = np.array([[1,0],[0,1]])
# iterate to get the state at timestep 3.
for i in range(3):
    x, P = iter_kf(x, P, u[i], z[i+1])
print("x:\n", x)
print("P:\n", P)
```

This results in the following estimate for our posterior distribution:

$$x_3 \sim \mathcal{N}\left(\begin{bmatrix}0.939\\0.166\end{bmatrix}, \begin{bmatrix}0.00829 & 0\\0 & 0.0168\end{bmatrix}\right)$$

We can see the top row (for the x-component) matches our result for part (a), as we would expect with the same inputs and measurements for that component of the state.

1.c 1-D motion

If we command changes in velocities instead of displacements ($u_t = \Delta v$), we may instead have the update rule $v_{t+1} = v_t + u_t +$velocity noise. Then position will be changed indirectly as $x_{t+1} = x_t + v_t +$position noise. For simplicity we assume here that changes in position occur before changes in velocity. This should read $x_{t+1} = x_t + v_t \cdot \Delta t +$ position noise, but in our case $\Delta t = 1$ second. Velocity cannot be directly observed, only the position as before.

The equations for this system are as follows. Our state holds both our position and velocity. Let μ_{vel} and σ_{vel} characterize the velocity noise, and μ_{pos} and σ_{pos} characterize the position noise.

$$\hat{X}^+_{t+1} = \begin{bmatrix} x \\ v_x \end{bmatrix}^+_{t+1} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v_x \end{bmatrix}_t + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} u_t \end{bmatrix} + \begin{bmatrix} \mu_{pos} \\ \mu_{vel} \end{bmatrix}$$

$$\hat{P}^+_{t+1} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{P}_{t+1} \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}^T + \begin{bmatrix} \sigma_{pos} & 0 \\ 0 & \sigma_{vel} \end{bmatrix}$$

$$\nu_{t+1} = z_{t+1} - \begin{bmatrix} 1 & 0 \end{bmatrix} \hat{X}^+_{t+1}$$

We can see that $F = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$, $G = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, and $H = \begin{bmatrix} 1 & 0 \end{bmatrix}$. This setup allows our position and velocity states to update simultaneously. The second half of the equations follow easily from this setup.

$$K_{t+1} = \hat{P}^+_{t+1} H^T \left(H \hat{P}^+_{t+1} H^T + \sigma_w^2 \right)^{-1}$$

$$\hat{X}_{t+1} = \hat{X}^+_{t+1} + K_{t+1} \cdot \nu_{t+1}$$

$$\hat{P}_{t+1} = (I_2 - K_{t+1} \cdot H) \hat{P}^+_{t+1}$$

I've translated this into the following python script.

```python
#!/usr/bin/env python3

# this code will perform a few iterations of a simple 2D Kalman filter
# using given parameters and velocity motion commands.
import numpy as np

## Global Variables ##
dt = 1
vel_noise_mu = 0; vel_noise_sig = 0.03
pos_noise_mu = 0; pos_noise_sig = 0.02
sig_w = 0.01
noise_mu = np.array([[pos_noise_mu,vel_noise_mu]]).T
V = np.array([[pos_noise_sig, 0],[0, vel_noise_sig]])
# W = np.array([[sig_w, 0],[0, sig_w]])
F = np.array([[1, dt], [0, 1]])
G = np.array([[0, 1]]).T
H = np.array([[1, 0]])
####

def iter_kf(X_0, P_0, u, z):
    """
    @param X_0, P_0 the prior (est. at time t)
    @param u the control input at time t
    @param z the measurement at time t+1
    @return X, P the posterior (est. at time t+1)
    """
    X_predicted = F @ X_0 + G*u + noise_mu
    P_predicted = F @ P_0 @ F.T + V
    innovation = z - H @ X_predicted
    K_gain = P_predicted @ H.T / (H @ P_predicted @ H.T + sig_w).item()
    X = X_predicted + K_gain * innovation
    P = (np.eye(2) - K_gain @ H) @ P_predicted
    return X, P

# index corresponds to timestep for u and z.
# control inputs.
u = [-0.2, 0.0, 0.1]
# measurements.
z = [None, 0.4, 0.9, 0.8]
# current state at t=0.
X = np.array([[0,0]]).T
P = np.array([[1,0],[0,1]])
# iterate to get the state at timestep 3.
for i in range(3):
    X, P = iter_kf(X, P, u[i], z[i+1])
print("x:\n", X)
print("P:\n", P)
```

This yields the following posterior distribution for $t = 3$.

$$\hat{X}_3 \sim \mathcal{N}\left(\begin{bmatrix}0.844\\0.227\end{bmatrix}, \begin{bmatrix}0.00920 & 0.00607\\0.00607 & 0.0505\end{bmatrix}\right)$$

1.d 2-d displacement w/ nonlinear measurements

We return to the robot on the 2D-plane in part (b), which is commanded by x-y displacements. Now we suppose that rather than measuring the position in each axis independently, we can only measure the squared distance to the robot, $z = x^2 + y^2 +$ observation noise.

Our motion model and control commands will be the same as in part (b), but our H matrix used to compute the innovation and Kalman gain will change.

$$\hat{X}_{t+1}^{+} = \hat{X}_t + u_t + \hat{\mu}_v$$
$$\hat{P}_{t+1}^{+} = \hat{P}_{t+1} + V$$

We can see that $F = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $G = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. We are not given a noise mean, so we must assume $\hat{\mu}_v = 0$. H is more complicated, since the relationship between the measurements and state is not a linear combination. As such, we must look to the EKF formulation, and use jacobian matrices to complete this task. (We could have done this for the prediction step as well, but it would give the same result, since there is no non-linearity present.

$$H_x = \frac{\partial h}{\partial \hat{X}} = \begin{bmatrix} \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x & 2y \end{bmatrix}$$
$$H_w = \frac{\partial h}{\partial \hat{w}} = \begin{bmatrix} \frac{\partial h}{\partial w} \end{bmatrix} = \begin{bmatrix} 1 \end{bmatrix}$$

We can then use these jacobians in our procedure, and will need to recompute them on each iteration to account for changing dependencies, i.e., the position. Since $H_w = 1$, I will not show it in the equations. Additionally, we are only given σ_w^2, so I must assume the noise mean $\hat{\mu}_w = 0$.

$$\nu_{t+1} = z_{t+1} - ||\hat{X}_{t+1}^{+}|| - \hat{\mu}_w$$
$$K_{t+1} = \hat{P}_{t+1}^{+} H_x^T \left(H_x \hat{P}_{t+1}^{+} H_x^T + \sigma_w^2 \right)^{-1}$$
$$\hat{X}_{t+1} = \hat{X}_{t+1}^{+} + K_{t+1} \cdot \nu_{t+1}$$
$$\hat{P}_{t+1} = (I_2 - K_{t+1} \cdot H_x) \hat{P}_{t+1}^{+}$$

I coded up this behavior, as shown below, and found a posterior distribution at $t = 3$ of

$$\hat{X}_3 \sim \mathcal{N}\left(\begin{bmatrix} 0.962 \\ 0.272 \end{bmatrix}, \begin{bmatrix} 0.0109 & -0.0342 \\ -0.0343 & 0.142 \end{bmatrix} \right)$$

```python
#!/usr/bin/env python3
import numpy as np

## Global Variables ##
sig_w = 0.01
V = np.array([[0.04, 0],[0, 0.09]])
F = np.eye(2); G = np.eye(2)
####

def iter_kf(X_0, P_0, u, z):
    """
    @param X_0, P_0 the prior (est. at time t)
    @param u the control input at time t
    @param z the measurement at time t+1
    @return X, P the posterior (est. at time t+1)
    """
    # prediction step.
    X_predicted = F @ X_0 + G @ u
    P_predicted = F @ P_0 @ F.T + V
    # compute jacobians for observation matrix. (H_w=1)
    H_x = np.array([[2*X_predicted[0,0], 2*X_predicted[1,0]]])
    # update step.
    innovation = z - (X_predicted[0,0]**2 + X_predicted[1,0]**2) # - noise?
    K_gain = P_predicted @ H_x.T / (H_x @ P_predicted @ H_x.T + sig_w).item()
    X = X_predicted + K_gain * innovation
    P = (np.eye(2) - K_gain @ H_x) @ P_predicted
    return X, P

# index corresponds to timestep for u and z.
# control inputs.
u = [np.array([[-0.5,0.3]]).T,
    np.array([[1.2,-0.6]]).T,
    np.array([[0.3,0.3]]).T]
# measurements.
z = [None, 0.6, 0.4, 1.0]
# current state at t=0.
X = np.array([[0,0]]).T
P = np.array([[1,0],[0,1]])
# iterate to get the state at timestep 3.
for i in range(3):
    X, P = iter_kf(X, P, u[i], z[i+1])
print("x:\n", X)
print("P:\n", P)
```

1.e Light-dark domain

The light-dark domain refers to an environment in which the robot may be incentivized to take a less direct path to the goal in order to better localize itself by moving into or remaining in a "light" region, determined by the x-coordinate.

This formulation is very similar to part (b), except now our observation noise is a function of the robot's x-position; this means we must recompute W on each loop iteration. We can directly observe the x-y position

for our measurements, as in part (b). The KF equations for this setup follow, and exactly match part (b), with the addition of a step to calculate the new W.

$$\hat{x}_1^+ = \hat{x}_0 + u_0$$
$$\hat{P}_1^+ = \hat{P}_0 + V$$
$$\nu_1 = z_1 - \hat{x}_1^+$$
$$W = \begin{bmatrix} \sigma_w^2(x) & 0 \\ 0 & \sigma_w^2(x) \end{bmatrix}, \text{ where } \sigma_w^2(x) = \frac{1}{2}(5-x)^2 + 0.01$$
$$K_1 = \hat{P}_1^+(\hat{P}_1^+ + W)^{-1}$$
$$\hat{x}_1 = \hat{x}_1^+ + K_1 \cdot \nu_1$$
$$\hat{P}_1 = (1 - K_1)\hat{P}_1^+$$

In the W equation, we use the x-coordinate of our predicted state to recompute it each update cycle. I coded this up as usual, and obtained a posterior distribution of

$$\hat{X}_3 \sim \mathcal{N}\left(\begin{bmatrix} 0.286 \\ -0.541 \end{bmatrix}, \begin{bmatrix} 0.0795 & 0.0 \\ 0.0 & 0.0795 \end{bmatrix}\right)$$

```python
#!/usr/bin/env python3

import numpy as np
from numpy.linalg import inv

## Global Variables ##
V = np.array([[0.01, 0],[0, 0.01]])
F=np.eye(2); G=np.eye(2); H=np.eye(2)
####

def iter_kf(x_0, P_0, u, z):
    """
    @param x_0, P_0 the prior (est. at time t)
    @param u the control input at time t
    @param z the measurement at time t+1
    @return x, P the posterior (est. at time t+1)
    """
    # prediction step.
    x_predicted = F @ x_0 + G @ u
    P_predicted = F @ P_0 @ F.T + V
    # update step.
    sig_w = (1/2)*(5-x_predicted[0,0])**2 + 0.01
    W = np.array([[sig_w, 0],[0, sig_w]])
    innovation = z - H @ x_predicted
    K_gain = P_predicted @ H.T @ inv(H @ P_predicted @ H.T + W)
    x = x_predicted + K_gain @ innovation
    P = (np.eye(2) - K_gain @ H) @ P_predicted
    return x, P

# index corresponds to timestep for u and z.
# control inputs.
u = [np.array([[1.0,-1.0]]).T,
    np.array([[2.0,-1.0]]).T,
    np.array([[-5.0,0.0]]).T]
# measurements.
z = [None,
    np.array([[3.5,-1.0]]).T,
    np.array([[5.3,-0.5]]).T,
    np.array([[-2.0,0.0]]).T]
# current state at t=0.
x = np.array([[2,2]]).T
P = np.array([[5,0],[0,5]])
# iterate to get the state at timestep 3.
for i in range(3):
    x, P = iter_kf(x, P, u[i], z[i+1])
print("x:\n", x)
print("P:\n", P)
```

1.f Extra credit

Discuss a paper that uses Kalman filtering to track some aspect of COVID-19.

I have not done this part.

2 Programming Section

E0 - EKF Using Robotics Toolkit

Implementing the three EKFs in this section using the code in the textbook (slightly modified) yields the figures as expected. The independent localization and mapping EKFs each perform very well, but the SLAM case fails horribly when using our passed parameters instead of the defaults. Several figures were produced when running E0, and two are shown in the figure below. The failed SLAM will be shown in comparison to my implementation in part E3, in Figure 4. I'm fairly confident that my implementation is not causing it to fail unfairly, since the exact same code that produces a scrambled mess for a range of 4 is perfectly capable of performing with a range of 8, as we see in part E4.

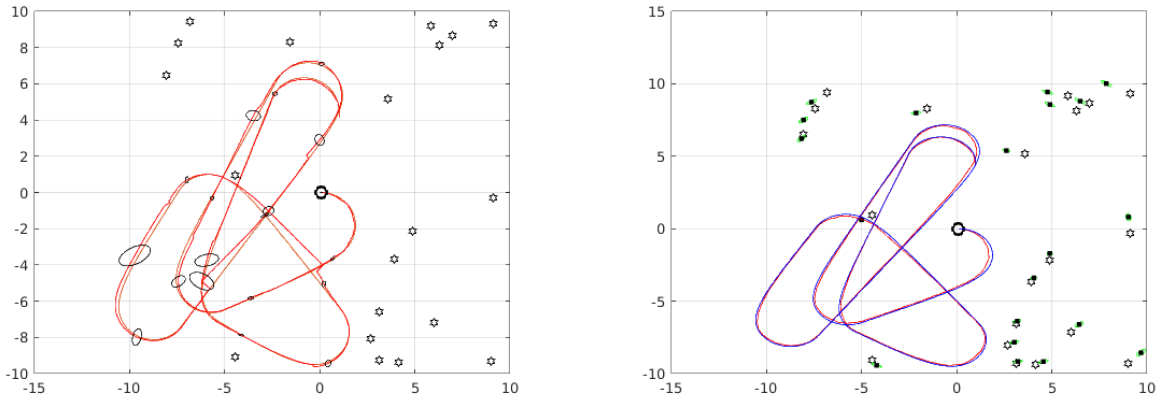

Figure 1: Some plots resulting from running E0.

E1 - EKF for Localization

We assume odometry is our control commands for distance and heading. Measurements involve a range and bearing to a certain landmark (known ID) with known global position since we have the map. We use the EKF to track the vehicle position and heading as the state.

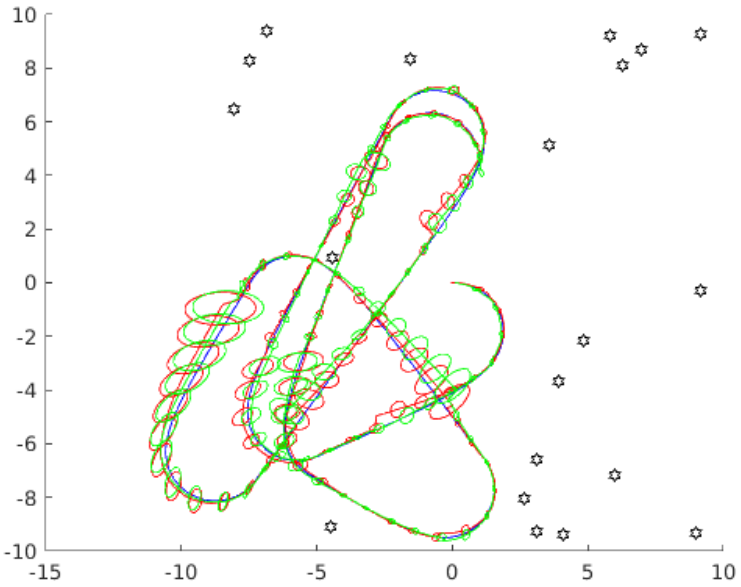

Figure 2: My EKF (red) using landmark detections for localization. Matches the true state (blue) and the Robotics Toolbox EKF implementation (green) very well.

E2 - EKF for Mapping

We assume odometry is exactly correct, so at all timesteps we have our true vehicle position. The map is not known, so using measurements of range and bearing to a landmark with known ID, our state is a list of the x,y positions of all seen landmarks.

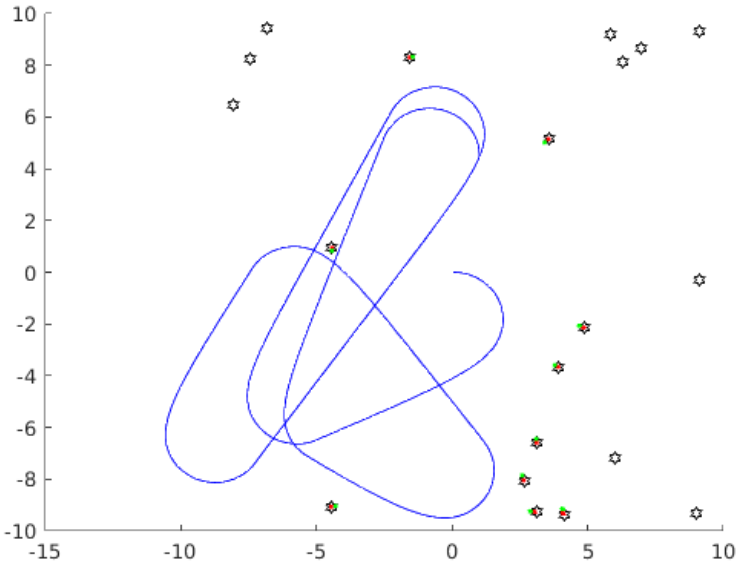

Figure 3: My EKF (red) using landmark detections for mapping. Matches the true map (black stars) and the Robotics Toolbox EKF implementation (green) very well.

E3 - EKF for SLAM

Now we don't have the true robot position or the true map, so we must track both the robot state (x,y,yaw) and the (x,y) position of all landmarks in our state. Here we will derive all Jacobians needed to perform EKF SLAM.

Our state will look like $\hat{x} = (x_v, y_v, \theta_v, x_1, y_1, ..., x_M, y_M)^T \in \mathbb{R}^{(2M+3)x1}$ for $M \in \mathbb{N}$ landmarks. Our covariance will look like

$$\hat{P} = \begin{bmatrix} \hat{P}_{vv} & \hat{P}_{vm} \\ \hat{P}_{vm}^T & \hat{P}_{mm} \end{bmatrix} \in \mathbb{R}^{(2M+3)\times(2M+3)}$$

where $\hat{P}_{vv}$ is the vehicle covariance, $\hat{P}_{mm}$ is the map covariance, and $\hat{P}_{vm}$ is the vehicle/map covariance.

First, the transition function. We assume landmark positions and covariance remain constant, while the robot position changes as it moves via the odometry commands $(\delta_d, \delta_\theta)^T$.

$$\hat{x}_{t+1} = f(\hat{x}_t, u_t) = \left[\begin{array}{c} x_{v,t} + (\delta_d + v_d)\cos\theta_{v,t} \\ y_{v,t} + (\delta_d + v_d)\sin\theta_{v,t} \\ \theta_{v,t} + \delta_\theta + v_\theta \\ \hline x_{1,t} \\ y_{1,t} \\ ... \\ x_{M,t} \\ y_{M,t} \end{array}\right]$$

Then the transition function Jacobian is:

$$F_x = \begin{bmatrix} F_{x,v} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & I_{2\times2} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & ... & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & I_{2\times2} \end{bmatrix},$$

where the submatrix for the vehicle position is

$$F_{x,v} = \begin{bmatrix} 1 & 0 & -\delta_d \sin\theta_v \\ 0 & 1 & \delta_d \cos\theta_v \\ 0 & 0 & 1 \end{bmatrix},$$

and the number of identity matrices on the diagonal equals M, the number of landmarks in the state. All zero matrices shown are the dimensions needed to fill out F_x. The transition noise Jacobian is

$$F_v = \begin{bmatrix} F_{v,v} \\ \mathbf{0}_{2\times2} \\ ... \\ \mathbf{0}_{2\times2} \end{bmatrix},$$

where the submatrix is

$$F_{v,v} = \begin{bmatrix} \cos\theta_v & 0 \\ \sin\theta_v & 0 \\ 0 & 1 \end{bmatrix},$$

and the number of zero matrices in F_v equals M, the number of landmarks in the state. We know that for the covariance prediction equation to work out, the noise matrix V must be 2×2.

$$\hat{P}^+_{t+1} = F_x \hat{P}_t F_x^T + F_v V F_v^T$$

Next, our observation function is

$$\hat{z} = h(x, p_i) = \begin{bmatrix} r + w_r \\ \arctan\frac{y_i - y_v}{x_i - x_v} - \theta_v + w_\beta \end{bmatrix},$$

where $r = \sqrt{(y_i - y_v)^2 + (x_i - x_v)^2}$. This only depends on the position of the newly measured landmark and the vehicle position. It is used for calculation of the innovation,

$$\nu = z^\# - \hat{z}$$

The observation Jacobians are then:

$$H_x = \begin{bmatrix} H_{x,v} & \mathbf{0} & \dots & H_{x,p_i} & \dots & \mathbf{0} \end{bmatrix}$$

where the submatrices for the vehicle and relevant landmark positions are

$$H_{x,v} = \begin{bmatrix} -\frac{x_i - x_v}{r} & -\frac{y_i - y_v}{r} & 0 \\ \frac{y_i - y_v}{r^2} & -\frac{x_i - x_v}{r^2} & -1 \end{bmatrix}$$

$$H_{x,p_i} = \begin{bmatrix} \frac{x_i - x_v}{r} & \frac{y_i - y_v}{r} \\ -\frac{y_i - y_v}{r^2} & \frac{x_i - x_v}{r^2} \end{bmatrix}$$

and the observation noise Jacobian is

$$H_w = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

For the Kalman gain equation to work out, the noise matrix W must be 2×2.

$$K = \hat{P}^+_{t+1} H_x^T \left(H_x \hat{P}^+_{t+1} H_x^T + H_w W H_w^T \right)^{-1}$$

We can see the portion being inverted will be 2×2, and the Kalman gain will be $(3 + 2M) \times 2$. This makes sense for our state update equations, since the innovation ν is 2×1.

$$\hat{x}_{t+1} = \hat{x}^+_{t+1} + K\nu$$
$$\hat{P}_{t+1} = \hat{P}^+_{t+1} - K H_x \hat{P}^+_{t+1}$$

Lastly, the insertion function:

$$g(x_v, z) = \begin{bmatrix} x_v + r\cos(\theta_v + \beta) \\ y_v + r\sin(\theta_v + \beta) \end{bmatrix}$$

$$\hat{x}_{t+1} = \begin{bmatrix} \hat{x}_t \\ g(x_v, z)_t \end{bmatrix}$$

So our insertion Jacobian is:

$$Y_z = \begin{bmatrix} I_{n\times n} & \mathbf{0}_{n\times 2} \\ \mathbf{0}_{2\times n} & G_z \end{bmatrix},$$

where $n = 3 + 2M$ (the state size) and the submatrix is:

$$G_z = \begin{bmatrix} \cos(\theta_v + \beta) & -r\sin(\theta_v + \beta) \\ \sin(\theta_v + \beta) & r\cos(\theta_v + \beta) \end{bmatrix}$$

These dimensions make sense when we look at our covariance update equation, where each term is a $(n + 2) \times (n + 2)$ matrix.

$$\hat{P}_{t+1} = Y_z \begin{bmatrix} \hat{P}_t & 0 \\ 0 & W \end{bmatrix} Y_z^T$$

We can then use our code from E1 and E2, as well as these derived Jacobians, to implement EKF SLAM in MATLAB. The results of this are shown in the following figure.

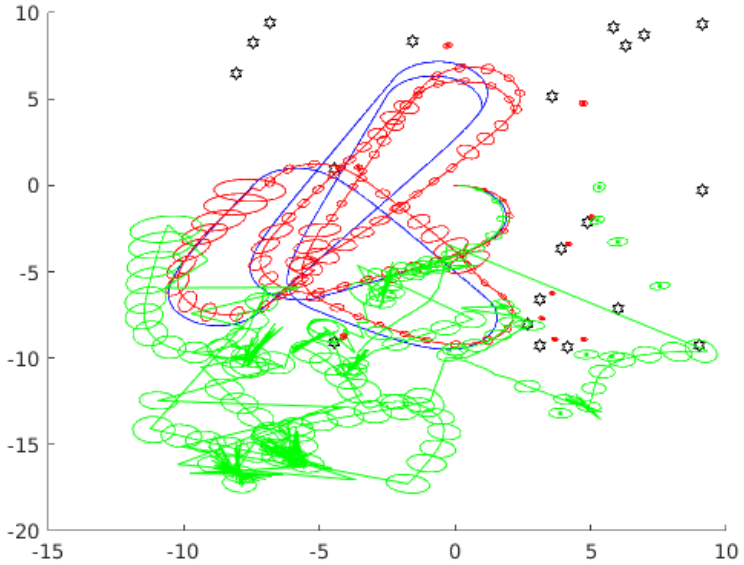

Figure 4: My EKF (red) using landmark detections for both localization and mapping. Matches the true pose and map (blue and black) fairly well. The Robotics Toolbox EKF implementation (green) fails dramatically.

E4

There are sections in the data collection during which no landmarks can be detected, meaning during these segments the uncertainty only grows. We hypothesize that increasing the sensor's maximum range will

improve its performance by reducing the amount of time for which there is no detection. We reuse our code from E0 to implement the toolbox's EKF SLAM, doubling the sensor range from 4 to 8.

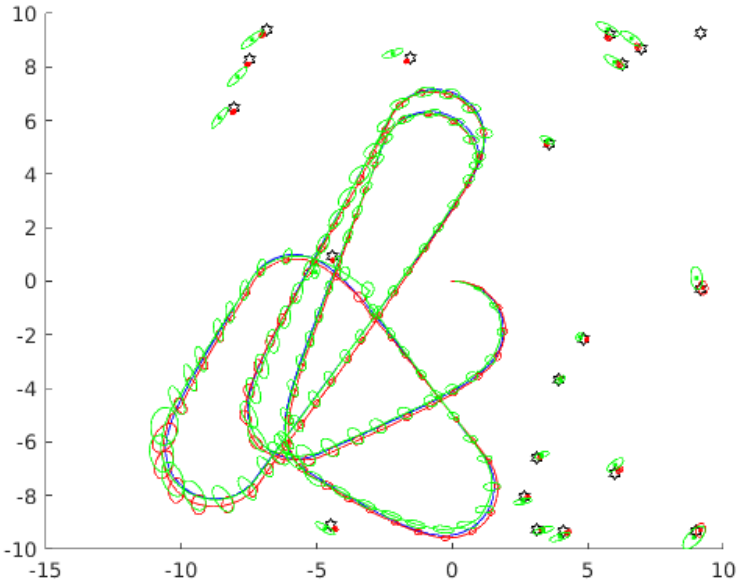

Figure 5: My EKF (red) using landmark detections for both localization and mapping, compared to the Robotics Toolbox implementation (green), both with an increased range of 8. Both match the true map very well and with low uncertainty. I would say my implementation outperforms the toolbox for mapping landmarks, while it does a better job of tracking the vehicle pose.