Homework 5

MacMillan, Kyle

November 26, 2018

Contents

List of Figures

1 Chapter 11

1.1 Problem 3

To complete this problem I created a class to generate a random map of any size you want up to 99. It generates a random number of obstacles of random size. Play around with it, it's fun. You can change the random numbers or just run the file multiple times. Code for this problem can be found here.

Demonstration is shown in Figure 1. This is an application of the WaveFront BFS algorithm. From there it goes on to find the shortest path as seen in Figure 2. The seed for that particular example is: `7635686187880284248`

Figure 1: WaveFront Distance Evaluation

Figure 2: WaveFront Shortest Path

I discussed the algorithm with Dr. McGough and he was in agreement that it didn't matter if you begin the wave at the start or goal. I reversed the point of origin for the WaveFront because it makes more sense in my brain. The result is the same so long as you start the WaveFront from one end and walk from the other end back to the WaveFront origin. It's more robust if done this way, that way if the goal is moving you don't have to redo the BFS, you just redo the navigation step.

2 Chapter 16

2.1 Problem 1

Given mesaurements of: $z_1 = 284$, $z_2 = 257$, and $z_3 = 295$ and standard deviations of $\sigma_1 = 10$, $\sigma_2 = 20$, and $\sigma_3 = 15$ we can setup variance as the square of the standard deviations:

1. 100
2. 400
3. 225

Given the variance we can apply the formulas from section 16.3.3 in the book to obtain an estimated state $\hat{x} = 282.90163934426226$. Code for this problem can be found here and in the code snippet 2.1.

```
# Given values
measured = np.array([284, 257, 295])
std = np.array([10, 20, 15])

# Calculate variance
var = std * std

# Sensor fusion to obtain x_hat
top = 0.0
bot = 0.0
for i in range(3):
    top += (measured[i] / var[i])
    bot += 1 / var[i]

# Estimated distance
x_hat = top / bot
```

2.2 Problem 2

Given 40 measurements at 2 meters we can calculate the mean:

A 2.23521735

B 1.86443904

C 2.33806001

That is how far off each senor averages from 2 meters. Each new sensor reading then has that mean subtracted from it to yield:

A 2.22255225

B 2.03233526

C 1.79719609

We can then apply the formula provided in the sensor fusion portion of the book for an expected distance, $\hat{x} = 2.057449909256987$ meters. The code for this can be found here and in the code snippet 2.2.

```python
# Calculate mean and standard deviation
mean = np.mean(dist_sens, dtype=np.float64, axis=0)
std = np.std(dist_sens, dtype=np.float64, axis=0)

# Reshape because it doesn't need to be 2D
np.reshape(mean, (1, 3))
np.reshape(std, (1, 3))

# Input for this step
new_sens = np.array([2.4577696, 1.8967743, 2.1352561]) + (2.0 - mean)

# Calculate variance
var = std * std

# Sensor fusion to obtain x_hat
top = 0.0
bot = 0.0
for i in range(3):
    top += (new_sens[i] / var[i])
    bot += 1 / var[i]

# Estimated distance
x_hat = top / bot
```

3 Chapter 17

3.1 Problem 2

The code for this problem can be found here.

The problem asked us to fuse three different measurements to determine an estimate of the covariance. To do that I first applied the same sensor fusion techniques utilized in Chapter 16, Problem 2. This sensor fusion led to:

$$\hat{x} = 10.557142857142857$$

$$\hat{y} = 18.18695652173913$$

$$x_{var} = 0.02857142857142857$$

$$y_{var} = 0.03260869565217391$$

Which can also be seen in Figure 3. We can also find the mean of x and y given the three initial measurements and can use them to find the covariance using:

$$\frac{\sum_{i=1}^{3} (\mu_x - x_i) * (\mu_y - y_i)}{3}$$

So putting it all together we end up with the covariance matrix:

$$P = \begin{bmatrix} 0.02857142857142857 & -0.1694444444444446 \\ -0.1694444444444446 & 0.03260869565217391 \end{bmatrix}$$

```
x_hat: 10.557142857142857
y_hat: 18.18695652173913
x_var: 0.02857142857142857
y_var: 0.03260869565217391
covar: -0.16944444444444462
```

Figure 3: Problem 17.2 Estimates

3.2 Problem 3

Lecture slide 58 is where I began for this problem.

Step 1: Discretize

This is accomplished with the Fundamental Theorem of Calculus.

$$y_n = \frac{x_{n+1} - x_n}{0.1}$$

$$\frac{y_{n+1} - y_n}{0.1} = -cos(x_n) + 0.5sin(t_n)$$

Step 2: Solve for $n + 1$

This is done with some simple algebra to move the terms around.

$$x_{n+1} = x_n + 0.1y_n$$

$$y_{n+1} = y_n - 0.1cos(x_n) + 0.05sin(t_n)$$

Step 3: Partial derivatives

$$F = \begin{bmatrix} \frac{\partial}{\partial x}(x_n + 0.1y_n) & \frac{\partial}{\partial y}(x_n + 0.1y_n) \\ \frac{\partial}{\partial x}(y_n - 0.1cos(x_n) + 0.05sin(t_n)) & \frac{\partial}{\partial y}(y_n - 0.1cos(x_n) + 0.05sin(t_n)) \end{bmatrix}$$

$$F = \begin{bmatrix} 1.0 & 0.1 \\ 0.1sin(x_n) & 1.0 \end{bmatrix}$$

Using those equations and matrices we can now apply the Extended Kalman Filter steps.

```python
# For each time step
for i in range(N):
    # Predict State
    x_hat0 = updateXhat0(x_hat0[0][0], x_hat0[0][1], t[i]) + r[i]

    # Predict estimate covariance
    P = updateP0(V, updateF(x_hat0[0][0]), P)

    # Optimal Kalman gain
    K = updateK(H, P, W)

    # Update state estimate
    z = x_hat0[0][0] + q[i]
    x_hat1 = updateXhat1(x_hat0, K, z)

    # Update estimate covariance
    P = updateP1(np.identity(2), K, H, P)
```

The code for Chapter 17, Problem 3 shows the steps we take to obtain a filtered prediction. Figure 4 shows the x component and Figure 5 shows the y component. The y component did not have an observation, so the prediction is not as close as the x component. The function calls just perform the necessary procedure to complete that step of the Extended Kalman Filter. Code for this problem can be found here.

Also to note is that the filter predicts y will follow in the general direction of x since it had no observations in y.

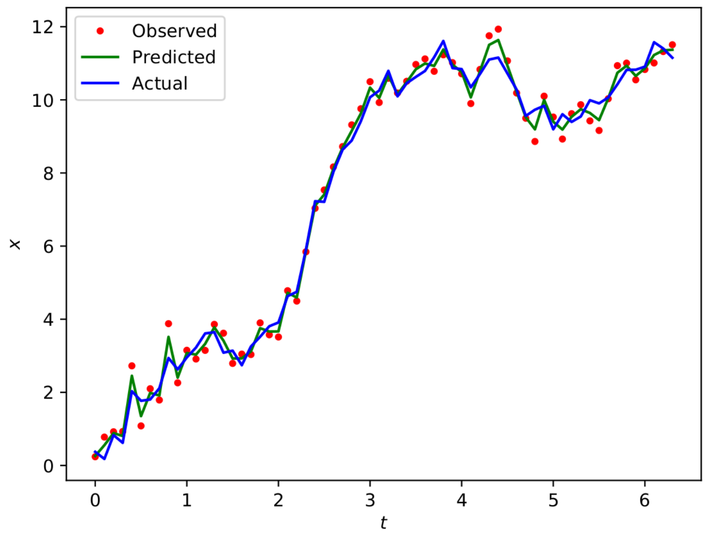

Figure 4: Problem 17.3 Kalman Filtering 1

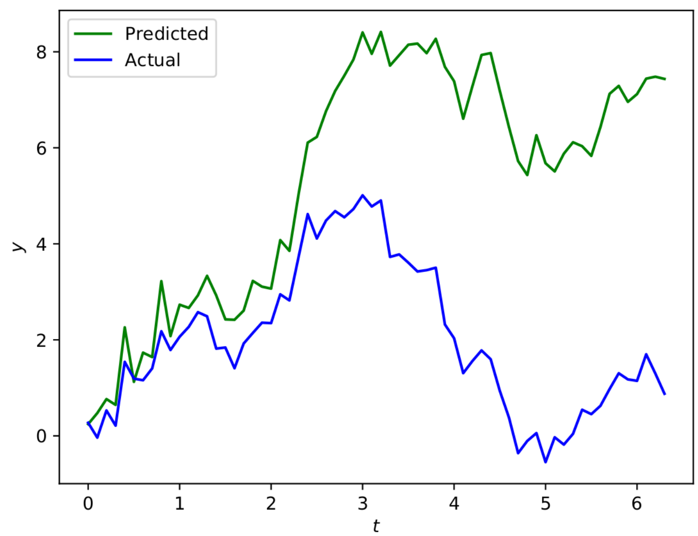

Figure 5: Problem 17.3 Kalman Filtering 2

3.3 Problem 4

3.3.1 Problem 4a

Figure 6 shows what the path would be with zero noise. Figure 7 shows the path with the V covariance matrix supplied and applied to numpy's multivariate random generator. As you can see the thing falls to pieces. I don't know why we would "convert to meters" but it appears the only way to get a legitimate plot is to use $V = V/100$, which can be seen in Figure 8.

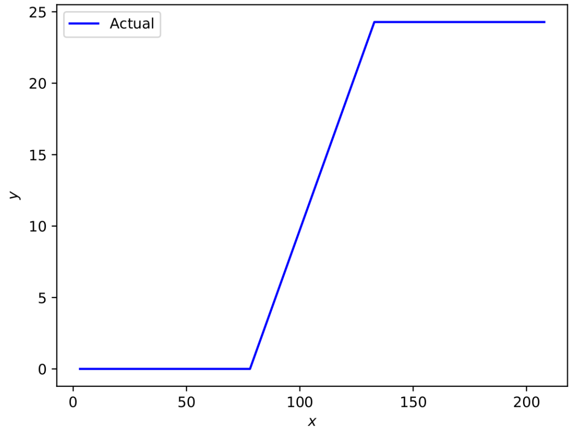

Figure 6: Problem 17.4 Robot Planned Path

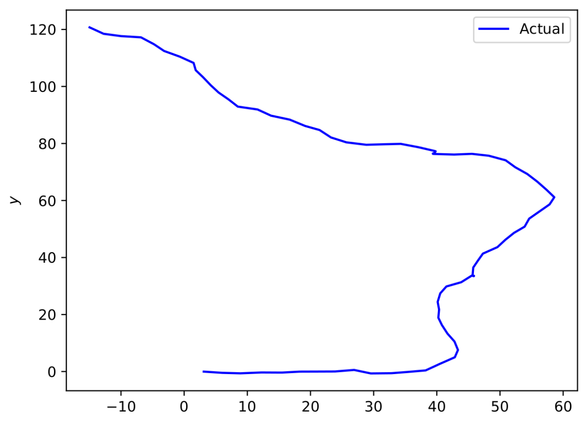

Figure 7: Robot planned path when V is applied

3.3.2 Problem 4b

3.3.3 Problem 4c

When $V = V/100$ we get the image in Figure 9

3.3.4 Problem 4d

Figure 10 shows xy on the vertical axis and time on the horizontal axis.

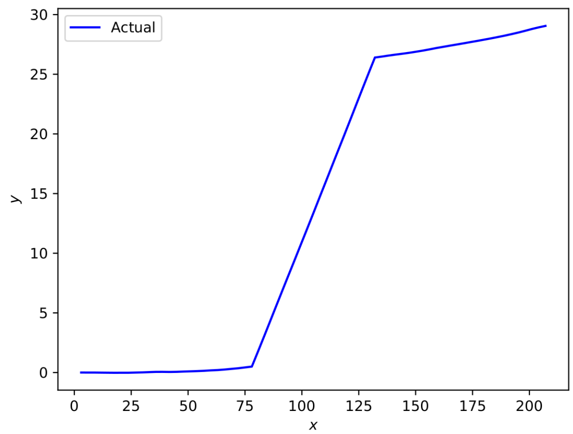

Figure 8: Robot planned path when V is divided by 100

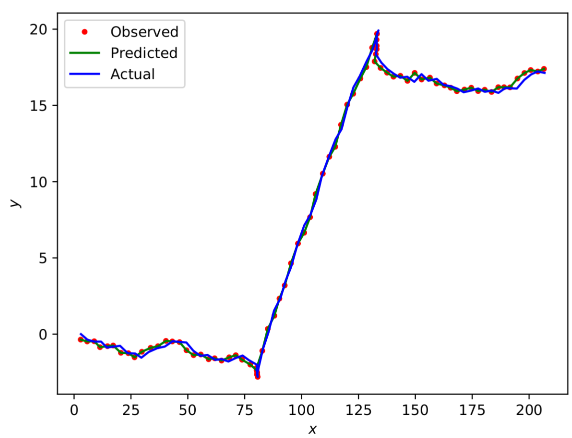

Figure 9: Problem 17.4 Robot x-y positions

3.3.5 Problem 4e

asdf

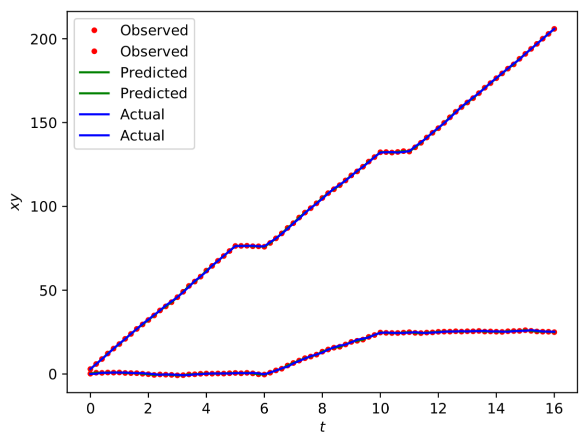

Figure 10: Problem 17.4 Robot x-y over time