ENMT482 Assignment 1 Guide

M.P. Hayes

1 Part A

This part of the assignment is about localising the position of a robot in 1-D using a number of sensors. The goal is to create sensor models, a motion model, and to use a Bayes filter to improve the estimate. The better the model, the better the estimation.

1.1 Sensor models

In general, for each sensor, you want a model of the form:

$$Z = h(x) + V(x). \tag{1}$$

1. Download the data files and example scripts from Learn.
2. Plot the sensor data using the file `calibration.csv` (see Python script `plot-calibration.py`).
3. Choose the sensors you wish to fuse (ignore the hard one to start with).
4. Fit a parametric model to the data for each sensor using parametric identification given measured pairs of data (x_n, z_n), see Figure 1. Let's say your model is:

   $$h(x) = a + bx + cx^2. \tag{2}$$

   The goal is to find the parameters a, b, and c that minimise the errors:

   $$v_n = z_n - h(x_n). \tag{3}$$

   Here's some example Python code for finding the unknown parameters:

   ```
   from scipy.optimize import curve_fit

   def model(x, a, b, c):

       return a + b * x + c * x * x


   # Load data x, z

   params, cov = curve_fit(model, x, z)

   zfit = model(x, *params)

   zerror = z - xfit
   ```

   The tricky aspect is dealing with outliers. One approach is to iteratively fit a model and then remove the obvious outliers. If you are not good at programming, just tweak your model parameters using trial-and-error until you get a good fit.
5. If you have a good model, the mean error is zero and the variance is minimized. I suggest plotting the error, v_n, as a function of x_n to see how good your model is. For example, see Figure 2.

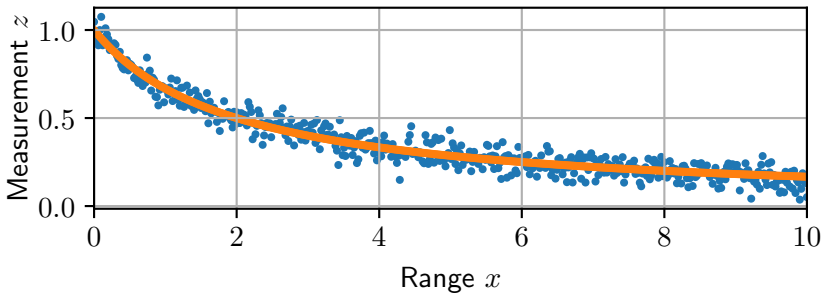

Figure 1: Non-linear sensor calibration data.

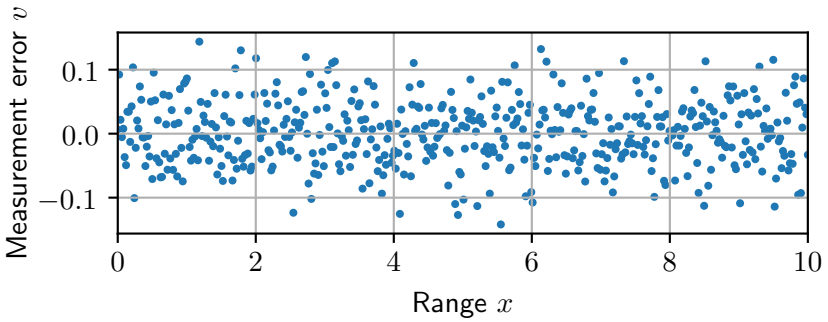

Figure 2: Error between measured data and model for non-linear sensor calibration data.

6. If you have outliers, you should remove them from your dataset. The simple approach is to ignore values with large errors. You may have to do this iteratively.

7. Determine how good each sensor is, i.e., what is its variance, see Figure 3. The tricky aspect is that some sensors have a variance that varies with x. If this is the case, you need to create a model for this, i.e., you need $\sigma_{V(x)}$. Note, if the variance changes slowly with x, a simple look-up table would suffice.

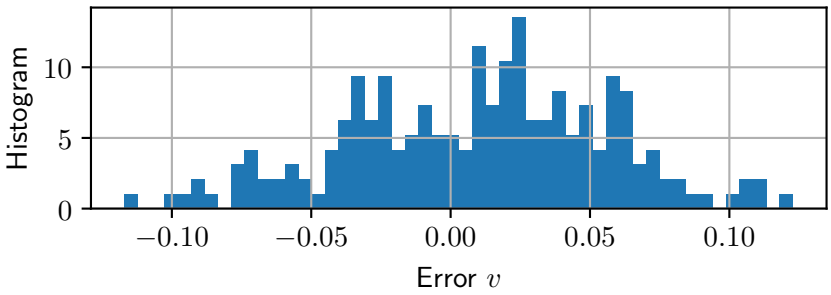

Figure 3: Histogram of errors between measured data and model for non-linear sensor calibration data in the range $0 \leq x \leq 1$.

1.2 Motion model

Here you need a model of the form:

$$X_n = X_{n-1} + g(X_{n-1}, u_{n-1}) + W_n, \tag{4}$$

where u_{n-1} is the commanded speed.

1. Plot the commanded speed and the estimated speed (using the distance data) in the training files, `training1.csv` and `training2.csv`, and ponder what is going on.

2. Determine a motion model, $g(x_{n-1}, u_{n-1})$. A crude model is simply

$$g(x_{n-1}, u_{n-1}) = u_{n-1}\Delta t, \tag{5}$$

   where u_{n-1} is the current commanded speed.

   A good model will have a zero mean error and minimise the variance of the errors. The process noise errors are given by

$$w_n = x_n - x_{n-1} - g(x_{n-1}, u_{n-1}). \tag{6}$$

3. Determine the process noise variance, σ_W^2, for your motion model.

1.3 Sensor fusion

Here you need a Bayes filter; an extended Kalman filter is easiest to start with. However, you will need to linearise your non-linear sensors around the current best estimate of the robot's position.

1. Predict the robot's position using the previous estimated position and your motion model.

2. Determine the variance of the predicted robot's position.

3. For each sensor, invert its model $h(x)$ so that given a measurement z you can have an estimate, $\hat{x}$, for x. However, some non-linear models are not easily invertible and so you may need to use interpolation to find $\hat{x}$ given z. Another approach is to use a bisection algorithm using $h(x) - z$ to find the root. Some models are invertible but have multiple solutions. In this case, choose the solution closest to the current estimate.

4. For each sensor, determine its noise variance $\sigma_V^2(x)$ at the current best estimate for x.

5. For each sensor, convert the noise variance $\sigma_V^2(x)$ into $\sigma_{\hat{X}}^2(x)$, where $\hat{X}$ is the estimator for X. This requires linearisation if your sensor model is non-linear (see the sensor fusion supplement in the lecture notes).

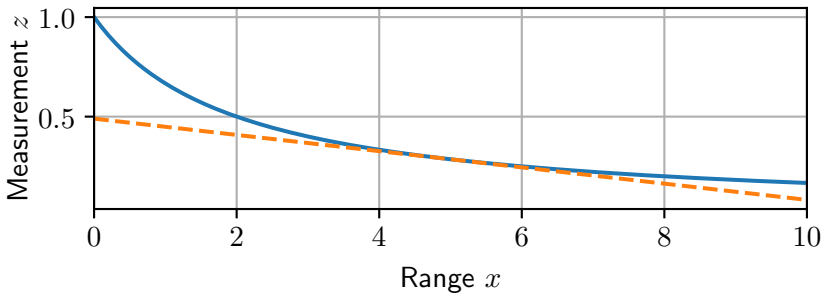

Figure 4: Non-linear sensor model linearised around $x = 5$.

6. Combine the estimates from each sensor using a BLUE (see lecture notes). To check that you have the correct weights, plot the weights for each sensor at each time-step.

7. Combine the prediction from the motion model with the estimates from the sensors using a BLUE. This can be done at the same time as the previous step.

8. Determine the variance of the BLUE estimator (see lecture notes).

9. Rinse and repeat.

Tips:

1. Test your filter with just the motion model.
2. Test your filter with the motion model and the best sensor and plot the BLUE weights. Do not worry about range varying variance; just use worst-case result. Single step each time-step and check the calculations.
3. Test your filter with the motion model and the two best sensors and plot the BLUE weights.

2 Part B

This part of the assignment is about localising the position of a robot in 2-D using fiducial markers (beacons) and a particle filter. These markers are sensed by a camera on a Turtlebot2 robot to estimate the local pose of the marker with respect to the robot.

The particle filter algorithm is written for you but you need to write motion and sensor models. If you are feeling clever, you might try adapting the number of particles and/or not using knowledge of the starting position.

2.1 Motion model

You can use either the velocity or odometry motion models. The latter has the advantage of decoupling the errors, see lecture notes.

To test your motion model, disable the sensor model (so that the particle weights do not change) and see if the particles move by the correct amount in the correct direction. A useful Python script is 'test-motion-model.py'.

Once the particles are moving correctly, add some random behaviour to the motion of each particle to mimic process noise. The amount of randomness depends on how good your motion model is. However, it is difficult to evaluate the process noise and I suggest that you tweak this by trial and error, starting with a small amount of noise.

2.2 Sensor model

To correctly implement the sensor model you will need to understand:

1. The difference between the robot and global (map) reference frames.
2. How to calculate the range and bearing of the beacons with respect to the robot given the estimated pose of the beacons.
3. How to calculate the range and bearing of the beacons with respect to each particle.
4. How to use the `arctan2` function.
5. How to determine the smallest angle between two vectors.

All these aspects are covered in the lecture notes.

Unfortunately, there is no calibration data for the fiducial marker sensor. I suggest modelling the sensor noise (in both range and bearing) as Gaussian random noise and choosing the standard deviation by trial and error. Note, the standard deviation will vary with the observed pose of the marker (it will be more accurate front-on than side-on).

2.3 Particle filter

The more particles you have, the better the estimate but the slower the computation.

Here's what I suggest you do:

1. Test motion model without adding noise to particles and disable sensor model. The particles should move in the correct direction.
2. Test motion model with added noise and disable sensor model. The particles should move in the correct direction but spread out.

3. Enable sensor model but with large standard deviations for the range and bearing errors. The particles should get closer together whenever a beacon is visible.

4. Reduce standard deviations in sensor model to get better tracking.