Sigma Point Filter and Particle Filter Problems
by Isaac Miller

I. Gaussian random vector generation. Your goal for this problem is to write a Matlab function capable of generating an n x 1 vector x drawn from a Gaussian with known n x 1 mean $\bar{x}$ and known n x n covariance matrix P_{xx}. We'll use the fact that a Gaussian random variable passed through a linear transformation is still Gaussian to help us out, by drawing a random vector from a zero-mean Gaussian with identity covariance and transforming it.

   a. Suppose the n x 1 vector $z \sim N(0, I)$ is Gaussian with zero mean and identity covariance. What are the mean and covariance of the linearly-transformed vector $x = A \cdot z + b$, where b is a known n x 1 vector and A is a known n x n matrix?

   b. What should the matrix A and the vector b be if we want the mean of x to be $\bar{x}$ and the covariance of x to be P_{xx}?

   c. Write a Matlab function that takes $\bar{x}$ and P_{xx} and generates a random variable x from the desired multivariate Gaussian. Check your answer by using your function to draw a large number of sample vectors: make sure the mean converges to an $\bar{x}$ that you pick and the covariance converges to a P_{xx} that you pick. *Hint:* the Matlab command rand(n, 1) generates an n x 1 random vector drawn from a zero-mean Gaussian with identity covariance.

II. The unscented transform. Your goal for this problem is to explore the effects of the linearized transform vs. the unscented transform for converting range and bearing measurements to Cartesian coordinates. The measurements will be generated from a radar whose errors are Gaussian and additive in range and bearing. The measurements are modeled as follows:

$$r = \bar{r} + w_r, w_r \sim N\left(0, \sigma_r^2\right)$$
$$\theta = \bar{\theta} + w_\theta, w_\theta \sim N\left(0, \sigma_\theta^2\right)$$

$$E\left[\begin{pmatrix} r \\ \theta \end{pmatrix}\right] = \begin{pmatrix} \bar{r} \\ \bar{\theta} \end{pmatrix}$$
$$E\left[\begin{pmatrix} r - \bar{r} \\ \theta - \bar{\theta} \end{pmatrix}\begin{pmatrix} r - \bar{r} & \theta - \bar{\theta} \end{pmatrix}\right] = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\theta^2 \end{bmatrix}$$

and the transformation to Cartesian coordinates is:

$$x = r \cdot \cos(\theta)$$
$$y = r \cdot \sin(\theta)$$

a. Linearize (Taylor expand and truncate after the 2nd term) x and y about the mean values $\bar{r}$ and $\bar{\theta}$. What are the mean and covariance of the vector $(x \quad y)^T$ under the linearization? Write a Matlab script to calculate the Cartesian mean and covariance matrix for the following values of $\bar{r}, \bar{\theta}, \sigma_r^2, \sigma_\theta^2$:

   i. $\bar{r} = 76, \bar{\theta} = -\frac{3\pi}{180}, \sigma_r^2 = 1^2, \sigma_\theta^2 = \left(\frac{\pi}{180}\right)^2$

   ii. $\bar{r} = 76, \bar{\theta} = -\frac{3\pi}{180}, \sigma_r^2 = 1^2, \sigma_\theta^2 = \left(\frac{15\pi}{180}\right)^2$

b. Now write a Matlab script to calculate the Cartesian mean and covariance for the values of $\bar{r}, \bar{\theta}, \sigma_r^2, \sigma_\theta^2$ given in part a) using the unscented transform. Use the following values for the unscented transform:

$$\alpha = 10^{-3}$$
$$\beta = 2$$
$$\kappa = 0$$

c. Use your Gaussian random vector generator to generate a large number of random vectors (a million or so) of ranges and bearings for each of the values of $\bar{r}, \bar{\theta}, \sigma_r^2, \sigma_\theta^2$ given in part (a). Transform each of these vectors to Cartesian coordinates using the full nonlinear transformation, and compute the sample Cartesian mean and covariance matrix over the sample points (this will be a good approximation to the true transformed mean and covariance). Compare these to the means and covariance matrices predicted by the linearization in part (a) and the unscented transform in part (b). Comment on the results.

III. The sigma point filter. Your goal for this problem is to build a sigma point filter to track a maneuvering car using range and bearing measurements generated from a laser rangefinder (LIDAR). We'll model the tracked car as a rectangle moving with a constant speed and heading, so the filter will have six states: x-position (m), y-position (m), speed (m/s), heading (rad.), length (m), and width (m). The x and y position components of the state will represent the center of the car's back axle, which we'll assume is the same as the middle of one of the edges of the car's rectangle. We'll model the car as moving with a random walk velocity and random walk heading:

$$\begin{aligned}
\dot{x} &= s \cdot \cos(\theta) + v_x \\
\dot{y} &= s \cdot \sin(\theta) + v_y \\
\dot{s} &= v_s \\
\dot{\theta} &= v_\theta \\
\dot{l} &= v_l \\
\dot{w} &= v_w
\end{aligned}$$

where x, y, s, θ, l, w are the elements of the car's state vector (x-position, y-position, speed, heading, length, and width), and $v_x, v_y, v_s, v_\theta, v_l, v_w$ are the elements of the process noise vector (x noise, y noise, speed noise, heading noise, length noise, and width noise). We'll be modeling all elements of the process noise vector as zero mean, white, and mutually uncorrelated, with process noise covariance matrix:

$$Q = \frac{diag\left(\begin{bmatrix} 0.25 & 0.25 & 3 & \frac{40\pi}{180} & 0.1 & 0.1 \end{bmatrix}\right)^2}{dt}$$

Notice some of our modeling choices: v_x, v_y are used to add "slop" to make sure the filter doesn't grow overconfident in its position estimate. The terms v_s, v_θ model the car's maneuvering; we're assuming the car typically accelerates and turns at rates less than 3 m/s^2 and 40^o/sec (which are both rather aggressive maneuvers). The terms v_l, v_w are used because the car's length and width are being estimated as constant parameters in the filter. If we were to eliminate these, the filter would eventually become perfectly confident in its length and width estimates… and it would effectively stop updating length and width. Notice also that we're using a discrete time approximation $Q \cong \frac{Q_c}{dt}$ of a continuous time white noise process Q_c. We'll be running the filter at 10 Hz, so we'll use $dt = 0.1$ sec.

Measurements for filter updating will be provided by a laser rangefinder. The LIDAR emits near-IR light and uses time-of-flight to calculate and return a set of ranges at bearings spaced half a degree apart. We'll assume the ground is perfectly flat and the LIDAR is horizontal, so that all the points it returns correspond to the car we want to track. We'll also assume the LIDAR itself is stationary for the duration of the data, and we'll do all our tracking in the LIDAR's coordinate frame, where the x-axis is defined as 0^o, positive bearings are counterclockwise, and the LIDAR itself sits at the origin.

Since the LIDAR doesn't exactly measure any of the car's states directly, we'll have to extract our own measurements from each frame of its data. We'll use a vector of three measurements to update the filter: minimum bearing (rad.), maximum bearing (rad.), and minimum range (m) (in that order) taken from the set of points that correspond to the car at each measurement update. The measurement vector will therefore be:

$$z = \begin{bmatrix} b_{\min} & b_{\max} & r_{\min} \end{bmatrix}^T$$

and we'll assume each measurement to be corrupted with mutually uncorrelated zero-mean additive white noise of covariance:

$$R = diag\left(\begin{bmatrix} \frac{2\pi}{180} & \frac{2\pi}{180} & 0.1 \end{bmatrix}\right)^2$$

in other words, we're modeling our angular measurements with about 2^o of noise at one standard dev., and our range measurements at about 10cm. Cheap commercial LIDARs can actually do a little better than that, but the extra noise helps account for the fact that it's discretized to half-degree increments in bearing.

Let's assume that other sensors close to the LIDAR give us an initial state estimate and covariance on the car:

$$x_{init} = \begin{bmatrix} 90 & 4.25 & 13 & \pi & 5 & 2 \end{bmatrix}^T$$
$$P_{init} = diag\left(\begin{bmatrix} 2 & 5 & 1 & \frac{\pi}{4} & 4 & 2 \end{bmatrix}\right)^2$$

Think of this initial state estimate and covariance as applying at time $t = -dt$ so that after propagation of the initial state you'll be ready to ingest the first measurements at $t = 0$.

Notice that the initial state estimate is fairly certain in the x-direction, speed, and width, but less certain in the y-direction and length. This type of behavior is typical of an automotive radar that returns decent ranges and speeds, but can't figure out heading, or lateral position very well. Also, since the target car is headed almost straight toward us, we have very little information about how long it actually is.

a. Write a sigma point filter to estimate the states of the target car. Use the following algorithmic steps as a guideline:

   i. Define and initialize all the vectors and matrices you'll use: $\hat{x}, P, Q, R$, as well as the parameters used by the sigma point filter: α, β, κ. Also load the LIDAR data stored in *problem3data.mat.*
   ii. Calculate the sigma points for the filter's prediction step. *Hint:* remember to calculate λ using the length of the appropriately-augmented state vector.
   iii. Perform a numerical integration on each sigma point to predict it to the time of the update. Use the supplied function *dyn_car.m* to evaluate the state derivative for the integration, and any numerical integration routine you like (e.g. Matlab's ode45).

iv. Use the predicted sigma points to calculate the predicted state $\bar{x}$ and its covariance matrix $\bar{P}$.

v. Extract the set of LIDAR returns for the ith sensor frame using the Matlab command *lidar(i).z*. The first column of this data is the set of ranges, and the second column is the set of bearings. Extract the minimum bearing, maximum bearing, and minimum range from this data to form the measurement vector z for the filter.

vi. Calculate the state and measurement sigma points for the filter's update step. *Hint:* remember to calculate λ using the length of the appropriately-augmented state vector. Use the supplied function *h_car.m* to calculate the measurement vector for each sigma point (this function performs geometric calculations on the estimated rectangle to obtain the measurement vector).

vii. Use the transformed sigma points to calculate the measurement covariance P_{zz} and also the state / measurement cross covariance P_{xz}.

viii. Perform the standard linear update to the LMMSE problem to obtain the updated state $\hat{x}$ and its covariance P.

ix. Repeat steps ii – viii for every frame of LIDAR data. You may use the function *plotcar.m* or any of its code to help visualize your estimate.

b. Compare your state estimate to the true state, stored in *problem3truth.mat*. In particular, consider the following points:

i. What effects, if any, do the values of α, β, κ have on the final state estimate? Use the typical values given in the lecture as a starting point for your exploration.

ii. What do you notice about the state estimates, in particular length and width, as the target car performs its different maneuvers?

iii. Why would it be difficult to implement this filter as an extended Kalman Filter?

iv. Why did we choose to use the bearing-bearing-range measurement instead of some other measurement vector?

IV. The (bootstrap) particle filter. Your goal in this problem is to build a particle filter to determine the location and heading of a "kidnapped" robot. The robot initially has no idea where he is, but he has a map telling him the location of five sonar beacons and he has a sonar unit that gives him (noisy) range measurements to the three beacons closest to him in ascending distance (in the simulated measurement data, sorting happens before addition of noise just to keep things simple). In addition, the robot has wheel encoders that give him (noisy) measurements of how much he's turned and how far he's traveled at a rate of 10 Hz. The noise on the robot's wheel encoders is mutually uncorrelated zero mean white noise with covariance:

$$Q = diag\left(\begin{bmatrix} 0.1 & \frac{5\pi}{180} \end{bmatrix}\right)^2$$

where the first element of the noise vector characterizes the noise on how far the robot has moved (meters), and the second element characterizes the noise on how far the robot has turned (rad.). Just to keep things clear, the robot performs two maneuvers at each time step: first he rotates in place a little, then he moves straight forward a little. After he's moved, the robot listens to his sonar, which gives him noisy ranges to the three beacons closest to him (but of course it doesn't tell him which beacons those are). Like the encoders, the noise on the sonar can be modeled as mutually uncorrelated zero mean white noise with covariance:

$$R = diag([1 \quad 1 \quad 1])^2$$

i.e. each of the three sonar measurements is typically accurate to about a meter.

a. Write a particle filter to determine the poor robot's position and heading. The state vector for your estimator should be $[x \quad y \quad \theta]^T$, where x is the robot's x-coordinate in the map (m), y is the robot's y-coordinate (m), and θ is the robot's heading (measured counterclockwise from the positive x direction). Use 1000 particles in your filter, and resample the filter when the effective number of particles falls below 500. Use the following algorithmic steps as a guideline:

    i. Load all the encoder and sonar data from the file *problem4data.mat*. The variables *minx, maxx, miny,* and *maxy* store the boundaries of the map, so you should use them to draw your initial set of particles from a uniform distribution over the entire map. Draw the heading dimension of your initial particle set from a uniform distribution on the range $[0, 2\pi]$. Also define the covariance matrices Q, R.
    ii. Extract the encoder commands for the ith time step from *encoder(i).u*. These commands apply for the interval whose endpoint is coincident with the ith sonar measurement *sonar(i).z.*[1] The encoder command vector is $u = [ds \quad d\theta]^T$, where $d\theta$ is how far the robot turns during this interval, and ds is how far he moves (in a straight line) after he's turned.
    iii. Draw sample values of the encoder process noise using your Gaussian random vector generator and use them with the encoder values to predict the particles forward to the time of the sonar measurement.
    iv. Extract the sonar measurement for the ith time step from *sonar(i).z*. This vector is a three element vector of the ranges to the three closest beacons, sorted in ascending order from smallest to largest.
    v. The variable *beacons* lists the locations of the five beacons: one beacon per row. Use *beacons* to compute the ranges to the three closest beacons for each particle.
    vi. Reweight each particle based on the measurement likelihood.

[1] Note that for our usual dynamics and measurement update formulas

$$x(k) = f[k-1, x(k-1), u(k-1), v(k-1)]$$
$$z(k) = h[k, x(k)] + w(k)$$

this arrangement of the encoder and measurement data implies that you will use *encoder(k).u* for $u(k-1)$ and *sonar(k).z* for $z(k)$.

vii. Calculate the effective number of particles after the reweighting, and resample the particles if necessary.

viii. Repeat steps ii – vii for every frame of sonar data. You may want to compute the minimum mean squared error estimate and the mean squared error matrix to visualize your solution.

b. Compare your state estimate to the true state, stored in *problem4truth.mat*. In particular, consider the following points:

   i. Does your state estimate time history change much if you run the filter more than once? Why might that be good or bad? *Hint:* try running the filter a few times with only 100 particles.

   ii. Why do clusters of particles sometimes persist in incorrect locations on the map?

   iii. Why would it be difficult to implement this filter as an extended Kalman Filter?