UKF-SLAM Derivation

Kevin Robb

Contents

1 Unscented Kalman Filter (UKF) Motivation

The Extended Kalman Filter introduced Jacobian matrices to linearize the system about the current mean at each iteration, so that even nonlinear systems can be estimated, as we've done in the previous sections. However, the state is still represented as a Gaussian random variable, and the linearized propagation technique only captures the first-order approximation of the nonlinear dynamics. This is the main drawback of the EKF, and can cause performance to fall off and even lead to divergence.

A different approach is the UKF, which entails choosing multiple "sigma points" to represent the characteristics of the state distribution, applying the nonlinear system to these points, and estimating a new distribution from the resulting set of points. If we think of the initial distribution as an infinite collection of points that has a particular mean and covariance, this process is analogous to down-sampling; we extract a set of $2n + 1$ points that have the same mean and covariance as the larger set. This way, when we propagate this small, finite set of points through the nonlinearity, we can convert it back into a Gaussian distribution very effectively, preserving much more of the distribution's characteristics than the EKF's Jacobian methods. This is effectively accurate up to a third-order approximation of the system, where the EKF had only first-order accuracy with the same order of complexity.

This may sound similar to a Particle Filter or other form of Monte-Carlo Localization; the difference is that for MCL, we have no assumptions about the state distribution, and use many hundreds of particles for them to eventually converge on the true state without fear of particle depletion. The UKF can use a very small number of points, since we still assume the state itself can be represented as a Gaussian. We only apply propagation to these few vectors, rather than some massive swarm of particles, meaning the runtime is far quicker.

2 UKF State Definition

The core innovation of the UKF is to represent the state distribution (normally done with a mean vector $\boldsymbol{x}_t$ and a covariance matrix $\boldsymbol{P}_t$) as a set of representative points. We will then be able to easily apply our true nonlinear model to this set of points, after which we must fit a new "posterior" state distribution to them. This posterior distribution, as well as the covariance matrices computed in the Update Stage, utilizes a weighted averaging of the sigma points. Thus, we need to define our state in such a way that weighted averaging is valid.

Until now, our state has been of the form $\boldsymbol{x}_t = \begin{bmatrix} x_v & y_v & \theta_v & x_1 & y_1 & \cdots & x_M & y_M \end{bmatrix}^T$. The yaw term θ_v fails our criterion. To see this, recall that all angles are represented in radians in the range $[-\pi, \pi]$. If the vehicle is facing to the left, then, some sigma points may be tilted up slightly, while others may be tilted down slightly. The average of these points should face to the left, aligning with the vehicle; however, if we take the average of the angles, we get $\theta_v = ((\pi - \varepsilon) + (-\pi + \varepsilon))/2 \approx 0$, which faces to the right.

To solve this problem, we will instead use the complex vector representation of the angle in our state, for which averaging is consistent with our expectations.

$$\boldsymbol{x}_t = \begin{bmatrix} x_v & y_v & \theta_{Re} & \theta_{Im} & x_1 & y_1 & \cdots & x_M & y_M \end{bmatrix}^T$$

If we were performing this project in a 3-dimensional workspace, our state would contain the 4D quaternion representation of the orientation in addition to the vehicle's and landmarks' x, y, z positions.

3 Prediction Stage

3.a Computing Sigma Points

Given the "prior" n-dimensional state distribution is $\sim \mathcal{N}(\boldsymbol{x}_t, \boldsymbol{P}_t)$, we will compute a representative set of points and their weights. These will be the matrix $\mathcal{X} \in \mathbb{R}^{n \times (2n+1)}$ and vector $\mathcal{W}$. Each column is a $n \times 1$ vector in the same format as the state mean, and column $\mathcal{X}_i$ is the sigma vector whose corresponding weight is $\mathcal{W}_i$.

These columns are computed as the following:

$$\mathcal{X}_i = \begin{cases} \boldsymbol{x}_t & \text{for } i = 0 \\ \boldsymbol{x}_t + \left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)_i & \forall i \in [1, n] \\ \boldsymbol{x}_t - \left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)_{i-n} & \forall i \in [n+1, 2n] \end{cases}$$

where we choose $\mathcal{W}_0 \in [-1, 1]$ to set the spread of the sigma points' distribution. A lower value will keep points closer to the origin. For this project, I've chosen $\mathcal{W}_0 = 0.2$. The weight for each sigma point is

$$\mathcal{W}_i = \frac{1 - \mathcal{W}_0}{2n} \qquad \forall i \in [1, 2n].$$

Note that all weights must sum to 1.

These equations are independent of the contents of our state and of the dynamics of the system, so they can certainly be applied in any project that has been able to utilize the KF/EKF already.

3.b Taking the Covariance Matrix Square Root

Computing the $\left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{P}_t}\right)$ matrix requires special attention; a matrix only has a square root if it is symmetric and positive-definite, which our covariance matrix is close to, but likely insufficient. In this step, then, we use the Frobenius norm formulation to find the closest matrix to $\boldsymbol{P}_t$ which satisfies these requirements. We do this with the following procedure:

1. Find the best *symmetric* matrix.
$$\boldsymbol{Y} = (\boldsymbol{P}_t + \boldsymbol{P}_t^T)/2$$

2. Multiply by the inner term for sigma points calculation.
$$\boldsymbol{Y} = \frac{n}{1 - \mathcal{W}_0} \cdot \boldsymbol{Y}$$

3. Find the eigen decomposition of $\boldsymbol{Y}$ to obtain $\boldsymbol{D}$, a $n \times 1$ vector of eigenvalues, and $\boldsymbol{Q}_v$, an $n \times n$ matrix of the corresponding eigenvectors.

4. Cap all eigenvalues to be $\geq \varepsilon$ for some small $\varepsilon > 0$.

5. Convert the vector of eigenvalues into a diagonal matrix, $\boldsymbol{D}_+$.

6. Compute $\boldsymbol{Z}$, the nearest symmetric, positive semi-definite matrix to $\boldsymbol{P}_t$.

$$\boldsymbol{Z} = Q_v \cdot \boldsymbol{D}_+ \cdot \boldsymbol{Q}_v^T$$

In C++, I've implemented this in the following function.

```
// find the nearest symmetric positive (semi)definite matrix to P_t using Frobenius Norm.
Eigen::MatrixXd UKF::nearestSPD() {
    // first compute nearest symmetric matrix.
    this->Y = 0.5 * (this->P_t + this->P_t.transpose());
    // multiply by inner coefficient for UKF.
    this->Y *= (2*this->M+3)/(1-this->W_0);
    // compute eigen decomposition of Y.
    Eigen::SelfAdjointEigenSolver<Eigen::MatrixXd> solver(this->Y);
    this->D = solver.eigenvalues();
    this->Qv = solver.eigenvectors();
    // cap eigenvalues to be strictly positive.
    this->Dplus = this->D.cwiseMax(0.00000001);
    // compute nearest positive semidefinite matrix.
    return this->Qv * this->Dplus.asDiagonal() * this->Qv.transpose();
}
```

Since we now have a symmetric, positive semi-definite covariance matrix, we can take its square root, which will also be a symmetric $n \times n$ matrix. Thus the terms such as

$$\left(\sqrt{\frac{n}{1-\mathcal{W}_0}\boldsymbol{Z}}\right)_i$$

can refer to either the ith row or ith column of the resulting square root matrix, since these are identical.

3.c Motion Model

Recall that our system dynamics are contained in the nonlinear function f (i.e., our motion model in the prediction step). We use the same SLAM model as with the EKF, and the same control command $\boldsymbol{u} = \begin{bmatrix} u_d & u_\theta \end{bmatrix}^T$ as input. The only difference is that, because we store the vehicle's yaw as a vector in the state, we must convert it back to an angle for the motion model calculations.

$$\theta_{v_t} = \arctan\left(\frac{\theta_{Im}}{\theta_{Re}}\right) \tag{1}$$

$$f(\boldsymbol{x}_t, \boldsymbol{u}_t, \boldsymbol{v}) = \begin{bmatrix} x_v + (u_d + v_d)\cos(\theta_{v_t}) \\ y_v + (u_d + v_d)\sin(\theta_{v_t}) \\ \cos(\theta_v + u_\theta + v_\theta) \\ \sin(\theta_v + u_\theta + v_\theta) \\ x_1 \\ y_1 \\ \vdots \\ x_M \\ y_M \end{bmatrix}.$$

Rather than computing the Jacobians $\boldsymbol{F}_x$ and $\boldsymbol{F}_v$, we directly propagate our sigma vectors through f to obtain $\boldsymbol{\mathcal{X}}_{pred} \in \mathbb{R}^{n\times(2n+1)}$, where each column is $\boldsymbol{\mathcal{X}}_{pred,i} = f(\boldsymbol{\mathcal{X}}_i)$. We can then compute the mean and covariance of this new distribution:

$$\boldsymbol{x}_{pred} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot \boldsymbol{\mathcal{X}}_{pred,i}$$

$$\boldsymbol{P}_{pred} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (\boldsymbol{\mathcal{X}}_{pred,i} - \boldsymbol{x}_{pred}) \cdot (\boldsymbol{\mathcal{X}}_{pred,i} - \boldsymbol{x}_{pred})^T + \boldsymbol{V}$$

4 Update Stage

We also use the same measurements $\boldsymbol{z}_{id} = \begin{bmatrix} z_r & z_\beta \end{bmatrix}^T$ as in EKF-SLAM. For each landmark detection, we check if it's being tracked in our state; if not, this is the first detection, and it must be inserted into our state. We have a separate function for each mode. On one timestep, we could potentially have several of each type of detection; for simplicity, we perform all landmark updates first, and then perform all landmark insertions.

4.a Landmark Update

This step is used when the landmark detected is already being estimated in our state, or if we're running the UKF in localization-only mode (meaning we have access to the true landmark map).

The nonlinear observation model h estimates the measurement we'd expect if our current state prediction were correct. Here x_l, y_l is the position of this landmark in our state, and θ_v is computed according to Equation (1).

$$h(\boldsymbol{x}_t, \boldsymbol{w}) = \begin{bmatrix} r_{est} \\ \beta_{est} \end{bmatrix} = \begin{bmatrix} \sqrt{(x_l - x_v)^2 + (y_l - y_v)^2} + w_r \\ \arctan\left(\frac{y_l - y_v}{x_l - x_v}\right) - \theta_v + w_\beta \end{bmatrix}$$

Note that one of our measurements is an angle, and thus will have the same problem as the vehicle yaw when it comes to averaging our sigma points; however, splitting it into vector components as we did for the state is far less necessary, so for now we will keep it in radians.

We will run each propagated sigma vector through this sensing model to obtain $\boldsymbol{\mathcal{X}}_{zest} \in \mathbb{R}^{2\times(2n+1)}$, where each column is $\boldsymbol{\mathcal{X}}_{zest,i} = h(\boldsymbol{\mathcal{X}}_{pred,i})$.

We can then compute our overall measurement estimate

$$\boldsymbol{z}_{est} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot \boldsymbol{\mathcal{X}}_{zest,i}.$$

We then compute covariance matrices from these sigma points: the innovation covariance, denoted $\boldsymbol{S}$, and the cross-covariance between $\boldsymbol{x}_{pred}$ and $\boldsymbol{z}_{est}$, denoted $\boldsymbol{C}$.

$$\boldsymbol{S} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est}) \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est})^T + \boldsymbol{W}$$
$$\boldsymbol{C} = \sum_{i=0}^{2n} \mathcal{W}_i \cdot (f(\mathcal{X}_i) - \boldsymbol{x}_{pred}) \cdot (h(\mathcal{X}_i) - \boldsymbol{z}_{est})^T$$

We can then compute the Kalman Gain,

$$\boldsymbol{K} = \boldsymbol{C} \cdot \boldsymbol{S}^{-1}.$$

And now we can finish the update step by computing the posterior distribution.

$$\boldsymbol{x}_{pred} = \boldsymbol{x}_{pred} + \boldsymbol{K} \cdot (\boldsymbol{z} - \boldsymbol{z}_{est})$$
$$\boldsymbol{P}_{pred} = \boldsymbol{P}_{pred} - \boldsymbol{K} \cdot \boldsymbol{S} \cdot \boldsymbol{K}^T$$

We update the predicted state for each landmark, and after all updates/insertions have completed, we finalize the state estimate.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

4.b Landmark Insertion

Inserting a new landmark into the state is fairly simple; since this new measurement is the only information we have about the new landmark, we simply use it as our estimate. The process is much more straightforward than in EKF-SLAM, since no linearization needs to take place.

Compute the position of the landmark in the world frame (according to this new measurement) and append it to the end of the state mean.

$$g(\boldsymbol{x}, \boldsymbol{z}) = \begin{bmatrix} x_v + z_r \cos(\theta_v + z_\beta) \\ y_v + z_r \sin(\theta_v + z_\beta) \end{bmatrix}$$
$$\boldsymbol{x}_{pred} = \begin{bmatrix} \boldsymbol{x}_{pred} \\ g(\boldsymbol{x}, \boldsymbol{z}_{t+1}) \end{bmatrix}$$

Expand the covariance matrix with the sensing noise covariance, $\boldsymbol{W}$.

$$\boldsymbol{P}_{pred} = \begin{bmatrix} \boldsymbol{P}_{pred} & \boldsymbol{0}_{(3+2M)\times 2} \\ \boldsymbol{0}_{2\times(3+2M)} & \boldsymbol{W} \end{bmatrix}$$

Expand the process noise covariance matrix, since it's additive in the UKF and needs to be the same dimensions as $\boldsymbol{P}$. We don't want to inject noise into all landmark estimates on every prediction step, so the

noise covariance for all added rows/columns will be zeros.

$$\boldsymbol{V} = \begin{bmatrix} \boldsymbol{V} & \mathbf{0}_{(3+2M)\times 2} \\ \mathbf{0}_{2\times(3+2M)} & \mathbf{0}_{2\times 2} \end{bmatrix}$$

Lastly, increment the number of landmarks, $M = M + 1$.

After all updates/insertions have completed for this timestep, we finalize the state estimate.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_{pred}$$
$$\boldsymbol{P}_{t+1} = \boldsymbol{P}_{pred}$$

5 Demo

The UKF is very effective in my simulator in both localization mode and SLAM mode.

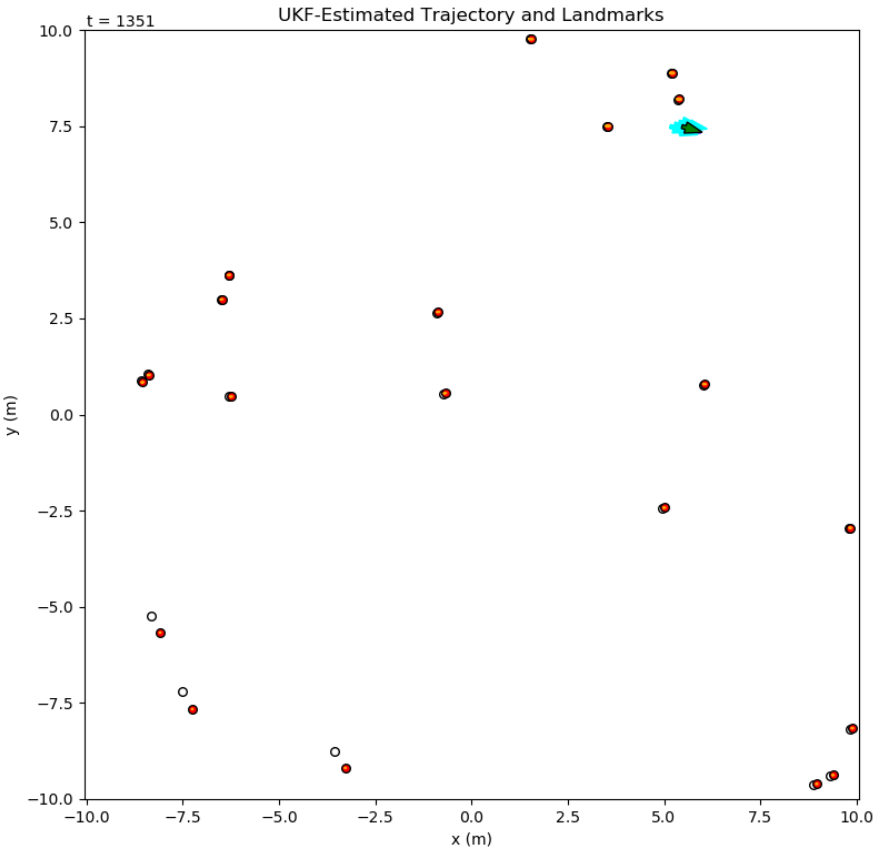

Figure 1: UKF performing online SLAM.