Probabilistic learning of dynamical systems

Introduction

Probabilistic model has the ability to represent and manipulate the uncertainty in dynamical systems, which can make predictions and decisions. There are no closed form solutions to these problems. We can approximately get the solutions using some statistical methods based on the measured data. Given the mathematical model and measurements, we can infer the unknown variables or parameters in the model. We are concerned with the problems of learning probabilistic models of dynamical systems from measured data, which are sometimes referred to as system identification problem. Based on the state-space model, the model can be expressed as follows.

$$x_t = f(x_{t-1}, u_t, v_t, \theta)$$
$$y_t = g(x_t, u_t, \theta) + e_t$$
$$x_0 \sim p(x_0 \mid \theta)$$
$$\theta \sim p(\theta)$$

We write the full joint distribution

$$p(x_{0:T}, \theta, y_{1:T})$$

Our goal is to estimate the unknown parameters

Probabilistic modeling of dynamical systems

The observed sequence y1:T, the conditional distribution can be

$$p(y_t \mid x_t, \theta) = p_{e_t}(y_t - g(x_t, \theta), \theta)$$

$$p(x_{0:T}, \theta, y_{1:T}) = p(y_{1:T} \mid x_{0:T}, \theta)\, p(x_{0:T} \mid \theta)\, p(\theta)$$

According to Markov property

$$p(y_{1:T} \mid x_{0:T}, \theta) = p(y_T \mid y_{1:T-1}, x_{0:T}, \theta)\, p(y_{1:T-1} \mid x_{0:T}, \theta)$$
$$= p(y_T \mid x_T, \theta)\, p(y_{1:T-1} \mid x_{0:T-1}, \theta)$$

$$p(x_{0:T} \mid \theta) = p(x_T \mid x_{T-1}, \theta)\, p(x_{0:T-1} \mid \theta)$$

Hence, we can get

$$p(x_{0:T}, \theta, y_{1:T}) = \left(\prod_{t=1}^{T} p(y_T \mid x_T, \theta) \right) \left(\prod_{t=1}^{T} p(x_T \mid x_{T-1}, \theta) \right) p(x_0 \mid \theta)\, p(\theta)$$

Compute the posterior given by sequence y

$$p(x_{0:T}, \theta \mid y_{1:T}) = p(x_{0:T} \mid \theta, y_{1:T})\, p(\theta \mid y_{1:T})$$

Because we are focused on the parameter inference, hence we can write the parameter inference

problem as

$$p(\theta \mid y_{1:T}) = \frac{p(y_{1:T} \mid \theta)p(\theta)}{\int p(y_{1:T} \mid \theta)p(\theta)d\theta}$$

Solving the parameter inference problem

The Monte Carlo idea

Monte Carlo integration provides approximate solutions to integrals of the form

$$c = E[h(x)] = \int_{\Xi} h(x)p(x)dx$$

From a Monte Carlo perspective, we can draw N random sample from probability distribution p and compute

$$\hat{c} = \frac{1}{N}\sum_{n=1}^{N} h(x^n)$$

The Markov Chain Monte Carlo idea

$$\frac{1}{M}\sum_{m=1}^{M} \varphi(\theta[m]) \to \int \varphi(\theta)p(\theta \mid y_{1:T})$$

To construct a Markov Chain, one way is to use the two-step method: 1) generate new candidates; 2) check if the candidate is accepted. The acceptance rate can be

$$\alpha = \min\left(1, \frac{p(y_{1:T} \mid \theta')p(\theta')}{p(y_{1:T} \mid \theta[m])p(\theta[m])} \frac{q(\theta[m] \mid \theta')}{q(\theta' \mid \theta[m])}\right)$$

Using unbiased estimates within Metropolis-Hastings

The MH Algorithm

$1: \textit{Initialization}: \textit{set } \theta[0] \textit{ and } \hat{z}[0]$
$2: \textit{for } m = 1 \textit{ to } M$
$3: \quad \textit{sample } \theta' \sim q(\theta \mid \theta[m-1])$
$4: \quad \textit{sample } \hat{z}' \sim \psi(z \mid \theta', y_{1:T})$
$5: \quad \textit{with probability}$
$\quad \alpha = \min\left(1, \frac{\hat{z}' p(\theta')}{\hat{z}[m-1]p(\theta[m-1])} \frac{q(\theta[m-1] \mid \theta')}{q(\theta' \mid \theta[m-1])}\right)$
$\quad \textit{set}\{\theta[m], \hat{z}[m]\} \leftarrow \{\theta', \hat{z}'\}, \textit{otherwise}$
$\quad \{\theta[m], \hat{z}[m]\} \leftarrow \{\theta[m-1], \hat{z}[m-1]\}$

This is based on the reject sampling which is given above. And now we derive the algorithm as follows.

Introduce a auxiliary variable

$$\psi(\theta, z \mid y_{1:T}) = p(\theta \mid y_{1:T})\psi(z \mid \theta, y_{1:T}) = \frac{\psi(z \mid \theta, y_{1:T})p(y_{1:T} \mid \theta)p(\theta)}{\int p(y_{1:T} \mid \theta)p(\theta)d\theta}$$

Replace the intractable likelihood with z, then we get

$$\psi(\theta, z \mid y_{1:T}) = \frac{z\psi(z \mid \theta, y_{1:T})p(\theta)}{\int p(y_{1:T} \mid \theta)p(\theta)d\theta}$$

Integrate the new joint distribution

$$\int \psi(\theta, z \mid y_{1:T})dz = \frac{p(\theta)}{p(y_{1:T})} \int z\psi(z \mid \theta, y_{1:T})dz$$

$$= \frac{p(\theta)}{p(y_{1:T})} p(y_{1:T} \mid \theta) = p(\theta \mid y_{1:T})$$

Computing an unbiased estimate of the likelihood using particle filter

$$p(y_{1:T} \mid \theta) = \int_x^{T+1} p(y_{1:T} \mid x_{0:T}, \theta)p(x_{0:T} \mid \theta)dx_{0:T}$$

Monte Carlo integration for the likelihood

1, generate state trajectory

2, compute the likelihood by integration

In order to reduce the computation, we rewrite the formula

$$p(y_{1:T} \mid \theta) = \prod_{t=1}^{T} \int_x p(y_t \mid x_t, \theta)p(x_t \mid y_{1:T-1}, \theta)dx_t$$

Introducing the particle filter

Draw N samples to approximate the distribution which gives

$$\hat{p}(x_t \mid y_t, \theta) = \frac{1}{N} \sum_{n=1}^{N} \delta(x_t^n)$$

$$\int_x p(y_t \mid x_t, \theta)p(x_t \mid y_{1:T-1}, \theta)dx_t = \frac{1}{N} \sum_{n=1}^{N} p(y_t \mid x_t^n, \theta)$$

First, by Bayes Theorem it follows that

$$p(x_t \mid y_{1:T}, \theta) \propto p(y_t \mid x_t, \theta)p(x_t \mid y_{1:T-1}, \theta)$$

By incorporating the state at time t+1 and marginalize over xt we have

$$p(x_{t+1} \mid y_{1:T}, \theta) = \int p(x_{t+1} \mid x_t, \theta)p(x_t \mid y_{1:T}, \theta)dx_t$$

Hence, the particle filter algorithm can be

1: *Initialization*:
2: $\quad$ *sample* $x_0 \sim p(x_0 \mid \theta)$ *and propagate* $x_1 \sim p(x_1 \mid x_0^n, \theta)$
3: $\quad$ *compute* $w_1^n = p(y_1 \mid x_1^n, \theta)$
4: *for* $t = 2$ *to* T *do*
5: $\quad\quad$ *resample* a_t^n *with* $p(a_t^n = j) \propto\sim w_{t-1}^n$ *and set* $\bar{x}_{t-1}^n = x_{t-1}^{a_t^n}$
6: $\quad\quad$ *sample* $x_t^n \sim p(x_t \mid \bar{x}_{t-1}^n, \theta)$
7: $\quad\quad$ *compute* $w_t^n \sim p(y_t \mid x_t^n, \theta)$
8: *end*

Hence, the z estimator can be calculated

$$\hat{z} = \prod_{t=1}^{T} \left[\frac{1}{N} \sum_{n=1}^{N} w_t^n \right]$$

Using the particle filter estimate within MH

1: *Initialization*: *set* $\theta[0]$ *and* $\hat{z}[0]$
2: *for* $m = 1$ *to* M
3: $\quad$ *sample* $\theta' \sim q(\theta \mid \theta[m-1])$
4: $\quad$ *sample* $\hat{z}' \sim \psi(z \mid \theta', y_{1:T})$ *by running particle filter*
5: $\quad$ *with probability*
6: $\quad\quad \alpha = \min\left(1, \frac{\hat{z}' p(\theta')}{\hat{z}[m-1] p(\theta[m-1])} \frac{q(\theta[m-1] \mid \theta')}{q(\theta' \mid \theta[m-1])}\right)$
7: $\quad\quad$ *sample* w_m *uniformly over* $[0, 1]$
8: $\quad\quad$ *if* $w_m < \alpha$
9: $\quad\quad\quad \{\theta[m], \hat{z}[m]\} \leftarrow \{\theta', \hat{z}'\}$
10: $\quad\quad$ *else*
11: $\quad\quad\quad \{\theta[m], \hat{z}[m]\} \leftarrow \{\theta[m-1], \hat{z}[m-1]\}$

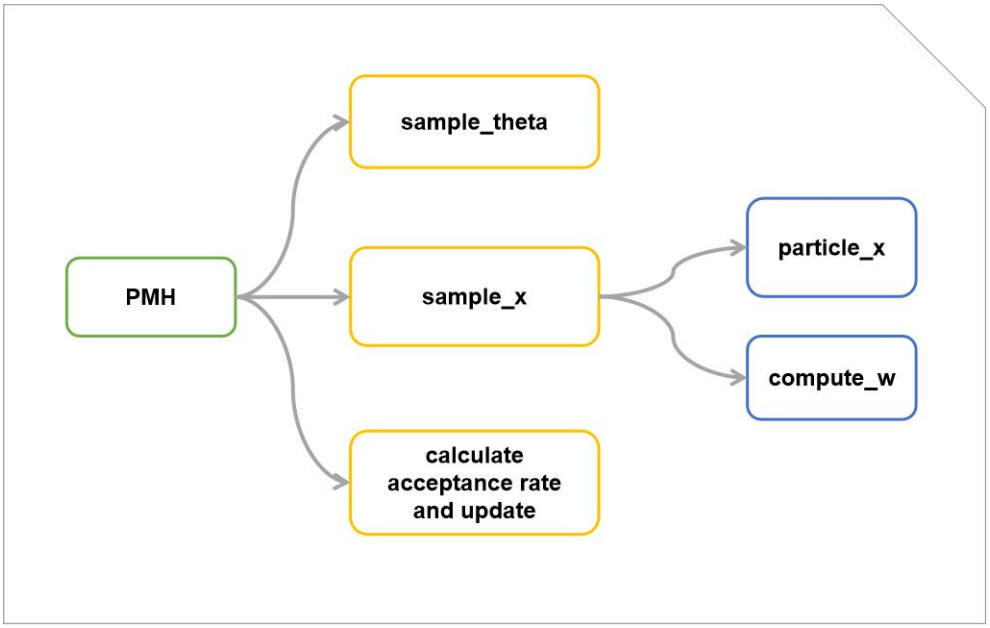
Reference

Thomas B. Schon *Probabilistic learning of nonlinear dynamical systems using sequential Monte Carlo*

Implementation

Based on the PMH mentioned above, we write the code step by step without optimizing the code, which is rough version. The code will be implemented on Pycharm with some built-in libraries. Numpy and Scipy.stats are mainly used.

The framework of the algorithm will be as follows.



<sample_theta>

```
def sample_theta(self, mean, cov):
    theta = st.multivariate_normal.rvs(mean=mean, cov=cov, size=10)
    theta_ = theta.mean(0)
    theta_cov_ = np.cov(theta.T)

    return theta_, theta_cov_
```

Here, we take all parameters as a joint normal distribution for convenience.

<particle_sampling>

```
def particle_sampling(self, x_param, matrix_param, cov_param):

    x_pred = matrix_param.dot(x_param.mean(0).T)
    x_cov_pred = matrix_param.dot(cov_param).dot(matrix_param.T)

    x_ = st.multivariate_normal.rvs(mean=x_pred, cov=x_cov_pred, size=self.sample_num_)

    return x_, x_cov_pred
```

Here, we use the mean of all the particles and the particles share same covariance with each other.

<compute_w>

```
def compute_w(self, y_param, x_param, matrix_param):

    e = y_param - matrix_param[0, 0] * x_param[:, 0] - matrix_param[0, 1] * x_param[:, 1]
    w_ = st.multivariate_normal.pdf(e, 0, 0.5)

    return w_
```

<sample_x>

```
def sample_x(self, theta, y):

    sample_num = self.sample_num_
    sample_time = len(y)

    w = np.zeros((sample_num, sample_time))
    x0 = np.random.rand(10, 2)
    x_cov = np.eye(2, 2)
    A = theta.reshape((2, 2))

    x, x_cov = self.particle_sampling(x0, A, x_cov)
    w[:, 0] = self.compute_w(y[0], x, A)

    for i in range(1, sample_time):
        # sample an
        # an = np.random.randint(0, sample_num, sample_num)
        an = w[:, i - 1] * (sample_num - 1)
        an = an.astype(np.int)
        # set x_an
        x_an = x[an]
        # calculate mean and covariance
        x_new, x_cov = self.particle_sampling(x_an, A, x_cov)
        w[:, i] = self.compute_w(y[i], x_new, A)
        x = x_new

    # calculate z
    z_ = 1
    for i in w.mean(0):
        z_ = z_ * i

    return z_
```

Here, we use a linear combination, which uses matrix multiplication as out model. Finally we calculate the z.

<PMH>

```
def pmh(self, y):

    # this is for theta distribution
    theta_mean = np.array([0, 0, 0, 0])
    theta_cov = np.eye(4, 4)
    # this is for theta_m
```

```
    theta_m_mean = np.random.randn(4)
    theta_m_cov = np.eye(4, 4)
    z0 = 0.5

    theta = 0
    for m in range(self.iter_m_):

        theta_new, theta_cov_new = self.sample_theta(mean=theta_m_mean, cov=theta_m_cov)

        z_hat = self.sample_x(theta_new, y)

        z = z_hat / z0

        p = st.multivariate_normal.pdf(theta_new, mean=theta_mean, cov=theta_cov) / \
            st.multivariate_normal.pdf(theta_m_mean, mean=theta_mean, cov=theta_cov)

        q = st.multivariate_normal.pdf(theta_m_mean, mean=theta_new, cov=theta_cov_new) / \
            st.multivariate_normal.pdf(theta_new, mean=theta_m_mean, cov=theta_m_cov)

        a = min(1, z*p*q)
        u = st.uniform.rvs(0, 1)

        if a > u:
            theta_m_mean = theta_new
            theta_m_cov = theta_cov_new
            z0 = z_hat

        theta += theta_m_mean

    return theta_m_mean.reshape((2, 2)) / self.iter_m_
```

This is the whole algorithm. We compute the acceptance rate to update the parameters.

Problems

Some problems may occur during the computation due to divergence. This may come from the approximation error, initialization. Because of drawing random particles, it may lead to divergence if the number of the samples is small. If the number of the particles are large, the computation will be time-consuming. There is strong correlation between the mathematical model and measured data. All the parameters can be inferred based on the model and measurements.

Revisiting Kalman Filter

Given the probabilistic state space model in discrete time

$$x_k = f_k(x_{k-1}, w_{k-1})$$
$$y_k = h_k(x_k, v_n)$$

This has the form of a hidden Markov model because the state is hidden.

$$p(x_k \mid x_{1:k-1}, y_{1:k-1}) = p(x_k \mid x_{k-1})$$

The measurements are conditionally independent given states.

$$p(y_k \mid x_{1:k}, y_{1:k-1}) = p(y_k \mid x_k)$$

Our goal is to estimate the state based on measurements. That is

$$p(x_k \mid y_{1:k})$$

Hence, the state space model can be written down.

$$x_k \sim p(x_k \mid x_{k-1})$$
$$y_k \sim p(y_k \mid x_k)$$

Compute the state x_t posterior based on x_{t-1}.

$$p(x_k \mid y_{1:k}) = \eta p(y_k \mid x_k) \int p(x_k \mid x_{k-1}) p(x_{k-1} \mid y_{1:k-1}) dx_{k-1}$$

Conventional Kalman Filter

$$x_k = A_{k-1}x_{k-1} + B_{k-1}u_{k-1} + q_{k-1}$$
$$y_k = H_k x_k + r_k$$

Prediction step

$$\boxed{\begin{aligned} m_k^- &= A_{k-1}m_{k-1} \\ P_k^- &= A_{k-1}P_{k-1}A_{k-1}^T + Q_{k-1} \end{aligned}}$$

Update step

$$\boxed{\begin{aligned} v_k &= y_k - H_k m_k^- \\ S_k &= H_k P_k^- H_k^T + R_k \\ K_k &= P_k^- H_k^T S_k^{-1} \\ m_k &= m_k^- + K_k v_k \\ P_k &= P_k^- - K_k S_k K_k^T \end{aligned}}$$

There are some variants of Kalman Filter, here we try to initialize the Kalman Filter using one method. The algorithm will be based on the steps mentioned above. Here, note that we use RungeKutta to solve the ordinary differential equations and add some noise in position as measurements. The specific RungeKutta methods will come from *Numerical Methods for Engineers* -- Steven C. Chapra. We use Matlab to demonstrate the implementation of the algorithm. The whole framework is very straightforward. There are two steps: Prediction and Update. We use a spring-mass-damper system.

$$\frac{dr}{dt} = v$$
$$\frac{dv}{dt} = a - 2\zeta\omega v - \omega^2 x$$
$$a = \frac{F}{m}$$
$$2\zeta\omega = \frac{c}{m}$$
$$\omega^2 = \frac{k}{m}$$

<Model>

```
function xDot = RHSOscillator(x, d )
if( nargin <1)
    xDot = struct('a',0,'omega',0.1,'zeta',0);
    return
end
xDot = [x(2); d.a-2*d.zeta*d.omega*x(2)-d.omega^2*x(1)];
```

<Initialization>

```
function d = KFInitialize( type, varargin )
% Default data structures
switch lower(type)
    case 'ukf'
        d = struct( 'm',[],'alpha',1, 'kappa',0,'beta',2, 'dT',0,...
            'p',[],'q',[],'f','','fData',[], 'hData',[],'hFun','','t',0);
    case 'kf'
        d = struct( 'm',[],'a',[],'b',[],'u',[],'h',[],'p',[],...
            'q',[],'r',[], 'y',[]);
    case 'ekf'
        d =
struct( 'm',[],'x',[],'a',[],'b',[],'u',[],'h',[],'hX',[],'hData',[],'fX',[],'p',
[],...
            'q',[],'r',[],'t',0, 'y',[],'v',[],'s',[],'k',[]);
    case 'ukfp'
        d = struct( 'm',[],'alpha',1, 'kappa',0,'beta',2, 'dT',0,...
            'p',[],'q',[],'f','','fData',[], 'hData',[],'hFun','','t',0,'eta',[]);
    otherwise
        error([type ' is not available']);
end
% Return the defaults
if( nargin == 1 )
    return
```

```
end
% Cycle through all the parameter pairs
for k = 1:2:length(varargin)
    switch lower(varargin{k})
        case 'a'
            d.a = varargin{k+1};
        case {'m' 'x'}
            d.m = varargin{k+1};
            d.x = varargin{k+1};
        case 'b'
            d.b = varargin{k+1};
        case 'u'
            d.u = varargin{k+1};
        case 'hx'
            d.hX = varargin{k+1};
        case 'fx'
            d.fX = varargin{k+1};
        case 'h'
            d.h = varargin{k+1};
        case 'hdata'
            d.hData = varargin{k+1};
        case 'hfun'
            d.hFun = varargin{k+1};
        case 'p'
            d.p = varargin{k+1};
        case 'q'
            d.q = varargin{k+1};
        case 'r'
            d.r = varargin{k+1};
        case 'f'
            d.f = varargin{k+1};
        case 'eta'
            d.eta = varargin{k+1};
        case 'alpha'
            d.alpha = varargin{k+1};
        case 'kappa'
            d.kappa = varargin{k+1};
        case 'beta'
            d.beta = varargin{k+1};
        case 'dt'
            d.dT = varargin{k+1};
        case 't'
            d.t = varargin{k+1};
        case 'fdata'
```

```
            d.fData = varargin{k+1};
        case 'nits'
            d.nIts = varargin{k+1};
        case 'kmeas'
            d.kMeas = varargin{k+1};
    end
end
```

<Prediction>

```
function d = KFPredict( d )
% The first path is if there is no input matrix b
if( isempty(d.b) )
    d.m = d.a*d.m;
else
    d.m = d.a*d.m + d.b*d.u;
end
d.p = d.a*d.p*d.a' + d.q;
```

<Update>

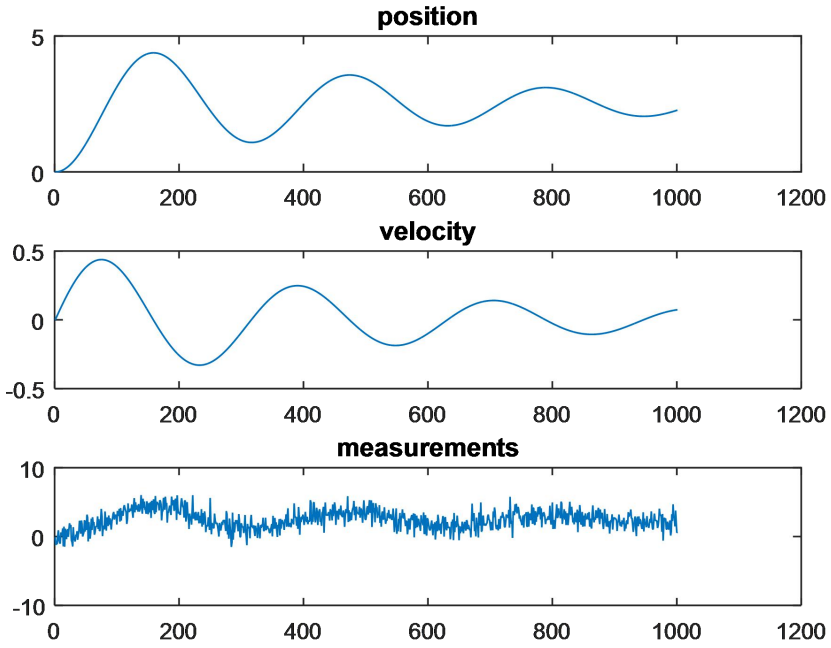
```
function d = KFUpdate( d )

s = d.h*d.p*d.h' + d.r; % Intermediate value
k = d.p*d.h'/s; % Kalman gain
v = d.y - d.h*d.m; % Residual
d.m = d.m + k*v; % Mean update
d.p = d.p - k*s*k'; % Covariance update
```

<Kalman_Filter>

```
%% KF Simulation
% Demostrate a Kalman Filter
%% Initialization
tEnd = 100.0; % Simulation end time (sec)
dT = 0.1; % Time step (sec)
d = RHSOscillator; % Get the default data structure
d.a = 0.1; % Disturbance acceleration
d.omega = 0.2; % Oscillator frequency
d.zeta = 0.1; % Damping ratio
x = [0;0]; % Initial state [position;velocity]
y1Sigma = 1; % 1 sigma position measurement noise

% xdot = a*x+b*u
a = [0, 1;-2*d.zeta*d.omega, -d.omega^2]; % Continuous time model
b = [0; 1]; % Continuous time input matrix
```

```
% x[k+1] = f*x[k] + g*u[k]
[f, g] = CToDZOH(a, b, dT); % Discrete time model

xE = [0.3; 0.1]; % Estimated initial state
q = [1e-6, 1e-6]; % Model noise covariance ;
% [1e-4 1e-4] is for low model noise test

dKF = KFInitialize('kf','m',xE,'a',f,'b',g,'h',[1 0],...
    'r',y1Sigma^2,'q',diag(q),'p',diag(xE.^2));

%% Simulation
nSim = floor(tEnd/dT) + 1;
xPlot = zeros(3,nSim);

for k = 1:nSim
    % Measurements (position)
    y = x(1) + y1Sigma*randn(1,1);

    % Update the Kalman Filter
    dKF.y = y;
    dKF = KFUpdate(dKF);

    % Plot storage
    xPlot(:,k) = [x;y];
    % Propagate (numerically integrate) the state equations
    x = RungeKutta(@RHSOscillator, x, dT, d);

    % Propagate the Kalman Filter
    dKF.u = d.a;
    dKF = KFPredict(dKF);
end

%% plot the results
t = 1:nSim;
name = ['position', 'velocity', 'measurements'];
plot_results(t, xPlot,name);
```

And the final results are



The extended Kalman Filter is used to solve nonlinear dynamical systems, which can be written as follows.

Prediction

$$m_k^- = f(m_{k-1})$$
$$P_k^- = F_x(m_{k-1})P_{k-1}F_x(m_{k-1})^T + Q_{k-1}$$

Update

$$v_k = y_k - h(m_k^-)$$
$$S_k = H_x(m_k^-)P_k^- H_x(m_k^-)^T + R_k$$
$$K_k = P_k^- H_x(m_k^-)^T S_k^{-1}$$
$$m_k = m_k^- + K_k v_k$$
$$P_k = P_k^- - K_k S_k K_k^T$$

The core idea here is to approximate the nonlinear equation by linearization.

Unscented Kalman Filter for State Estimation

With the UKF we work with the nonlinear dynamical and measurement equations directly. It is also known as a sigma point filter because it simultaneously maintains models one sigma fro the mean.

$$
\begin{aligned}
W_m^0 &= \frac{\lambda}{n+\lambda} \\
W_c^0 &= \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta \\
W_m^i &= \frac{1}{2(n+\lambda)} \; i = 1,...,2n \\
W_c^i &= \frac{1}{2(n+\lambda)} \; i = 1,...,2n
\end{aligned}
$$

Note that

$$
\begin{aligned}
W_m^i &= W_c^i \\
\lambda &= \alpha^2(n+\kappa) - n \\
c &= \lambda + n = \alpha^2(n+k)
\end{aligned}
$$

α: 0 for state estimation, 3 minus the number of states for parameter estimation

β: Determines spread of sigma points. Smaller means more closely spaced sigma points.

κ: Constant for prior knowledge. Set to 2 for Gaussian processes.

n is the order of the system. The weights can be put into matrix form

$$
\begin{aligned}
w_m &= \left[W_m^0 \cdots W_m^{2n}\right]^T \\
W &= \left(I - \left[w_m \cdots w_m\right]\right) \begin{bmatrix} W_c^0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & W_c^{2n} \end{bmatrix} \left(I - \left[w_m \cdots w_m\right]\right)^T
\end{aligned}
$$

Prediction

$$
\begin{aligned}
X_{k-1} &= \left[m_{k-1} \cdots m_{k-1}\right] + \sqrt{c}\begin{bmatrix} 0 & \sqrt{P_{k-1}} & \sqrt{P_{k-1}} \end{bmatrix} \\
\hat{X}_k &= f\left(X_{k-1}\right) \\
m_k^- &= \hat{X}_k w_m \\
P_k^- &= \hat{X}_k W \hat{X}_k^T + Q_{k-1}
\end{aligned}
$$

Update

$$
\begin{aligned}
X_k^- &= \left[m_k^- \cdots m_k^-\right] + \sqrt{c}\begin{bmatrix} 0 & \sqrt{P_k^-} & \sqrt{P_k^-} \end{bmatrix} \\
Y_k^- &= h\left(X_k^-\right) \\
\mu_k &= Y_k^- w_m \\
S_k &= Y_k^- W \left[Y_k^-\right]^T + R_k \\
C_k &= X_k^- W \left[Y_k^-\right]^T \\
K_k &= C_k S_k^{-1} \\
m_k &= m_k^- + K_k\left(y_k - \mu_k\right) \\
P_k &= P_k^- - K_k S_k K_k^T
\end{aligned}
$$

As we have taken a close look at the Kalman filter, we are now implementing the UKF algorithm to see how it works for spring-mass-damper system. The framework of the Kalman Filter. The key idea is to predict and update. The nonlinear equation will be used. The initialization is the same as Kalman Filter. We still use RungeKutta Methods to solve the ordinary differential equation.

<f_function>

```
function xDot = RHSOscillator(x, d )
if( nargin <1)
    xDot = struct('a',0,'omega',0.1,'zeta',0);
    return
end
xDot = [x(2); d.a-2*d.zeta*d.omega*x(2)-d.omega^2*x(1)];
```

<h_function>

```
function y = AngleMeasurement(x, d)
y = x(1) + d * rand(1, 1);
```

<Weight>

```
function d = UKFWeight( d )
% Compute the fundamental constants
n = length(d.m);
a2 = d.alpha^2;
d.lambda = a2*(n + d.kappa) - n;
nL = n + d.lambda;
wMP = 0.5*ones(1,2*n)/nL;
d.wM = [d.lambda/nL, wMP]';
d.wC = [d.lambda/nL+(1-a2+d.beta), wMP];
d.c = sqrt(nL);
% Build the matrix
f = eye(2*n+1) - repmat(d.wM,1,2*n+1);
d.w = f*diag(d.wC)*f';
```

<Prediction>

```
function d = UKFPredict( d )
pS = chol(d.p)'; % square root of covariance
nS = length(d.m);
nSig = 2*nS + 1;
mM = repmat(d.m,1,nSig);
x = mM + d.c*[zeros(nS,1) pS -pS];
xH = Propagate( x, d );
d.m = xH*d.wM;
d.p = xH*d.w*xH' + d.q;
```

```
d.p = 0.5*(d.p + d.p'); % Force symmetry

%% Propagate each sigma point state vector
function x = Propagate( x, d )
for j = 1:size(x,2)
    x(:,j) = RungeKutta(d.f, x(:, j), d.dT, d.fData);
end
```

<Update>

```
function d = UKFUpdate( d )
% Get the sigma points
pS = d.c*chol(d.p)';
nS = length(d.m);
nSig = 2*nS + 1;
mM = repmat(d.m,1,nSig);
x = mM + [zeros(nS,1) pS -pS];
[y, r] = Measurement( x, d );
mu = y*d.wM;
s =y*d.w*y' + r;
c =x*d.w*y';
k = c/s;
d.v = d.y - mu;
d.m = d.m + k*d.v;
d.p = d.p - k*s*k';

%% Measurement estimates from the sigma points
function [y, r] = Measurement( x, d )
nSigma = size(x,2);
% Create the arrays
lR = length(d.r);
y = zeros(lR,nSigma);
r = d.r;
for j = 1:nSigma
    f = feval(d.hFun, x(:,j), d.hData );
    iR = 1:lR;
    y(iR,j) = f;
end
```

<Unscented_Kalman_Filter>

```
%% UKFSim
% Demonstrate an Unscented Kalman Filter.
%% Initialize
nSim = 5000; % Simulation steps
```

```
dT = 0.1; % Time step (sec)

d = RHSOscillator; % Get the default data structure
d.a = 0.1; % Disturbance acceleration
d.omega = 0.1;
d.zeta = 0.1; % Damping ratio
x = [0;0]; % Initial state [position;velocity]
y1Sigma = 1; % 1 sigma measurement noise
dMeas.baseline = 10; % Distance of sensor from start
xE = [0; 0]; % Estimated initial state

q = diag([0.1 0.001]);
p = diag([0.01 0.0001]);

dKF = KFInitialize( 'ukf','m',xE,'f',@RHSOscillator,'fData',d,...
    'r',y1Sigma^2,'q',q,'p',p,'hFun',@AngleMeasurement,'hData',y1Sigma,'dT',dT);

dKF = UKFWeight( dKF );
%% Simulation
xPlot = zeros(3,nSim);
for k = 1:nSim
%     % Measurements
    y = AngleMeasurement( x, y1Sigma ) + y1Sigma*randn;
%     % Update the Kalman Filter
    dKF.y = y;
    dKF = UKFUpdate(dKF);
    % Plot storage
    xPlot(:,k) = [x;y];
%     % Propagate (numerically integrate) the state equations
    x = RungeKutta(@RHSOscillator, x, dT, d);
%     % Propagate the Kalman Filter
    dKF = UKFPredict(dKF);
end

%% plot the results
t = 1:nSim;
name = {'position', 'velocity', 'measurements'};
plot_results(t, xPlot,name);
```

So far we have gone through three kinds of Kalman Filters. They depend on noise covariance matrices Q and R. In practical application, we should determine the matrices more carefully.