TUTORIAL 4b

Kalman filtering

Course: Math 535 (http://www.math.wisc.edu/~roch/mmids/) - Mathematical Methods in Data Science (MMiDS)
Author: Sebastien Roch (http://www.math.wisc.edu/~roch/), Department of Mathematics, University of Wisconsin-Madison
Updated: Dec 3, 2020

1 Background: linear-Gaussian models

In this notebook, we illustrate the use of linear-Gaussian models for object tracking. We first give some background.

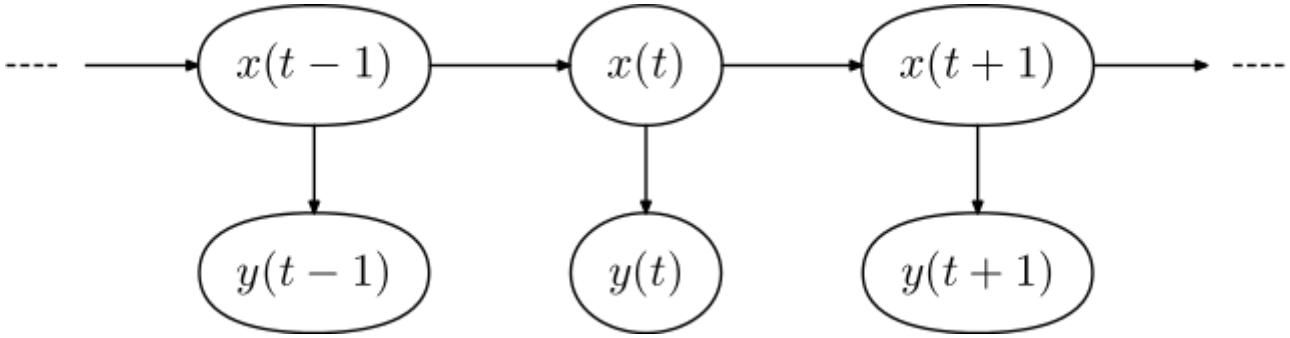
The model: We consider a Markov process $\{\mathbf{X}_t\}_{t=0}^T$ with state space $\mathcal{S} = \mathbb{R}^{d_0}$ of the following form

$$\mathbf{X}_{t+1} = F\,\mathbf{X}_t + \mathbf{W}_t$$

where the $\mathbf{W}_t$'s are i.i.d. $\mathcal{N}(\mathbf{0}, Q)$ and F and Q are known $d \times d$ matrices. We denote the initial state by $\mathbf{X}_0 = \mathbf{x}_0$. We assume that the process $\{\mathbf{X}_t\}_{t=1}^T$ is not observed, but rather that an auxiliary observed process $\{\mathbf{Y}_t\}_{t=1}^T$ with state space $\mathcal{S} = \mathbb{R}^d$ satisfies

$$\mathbf{Y}_t = H\,\mathbf{X}_t + \mathbf{V}_t$$

where the $\mathbf{V}_t$'s are i.i.d. $\mathcal{N}(\mathbf{0}, R)$ and $H \in \mathbb{R}^{d \times d_0}$ and $R \in \mathbb{R}^d$ are known matrices. Graphically, it can be represented as follows, where as before each variable is node and its conditional distribution depends only on its parent nodes.



(Source (https://commons.wikimedia.org/wiki/File:Hmm_temporal_bayesian_net.svg))

Our goal is to infer the unobserved states given the observed process. Specifically, we look at the filtering problem. Quoting Wikipedia (https://en.wikipedia.org/wiki/Hidden_Markov_model#Filtering):

> The task is to compute, given the model's parameters and a sequence of observations, the distribution over hidden states of the last latent variable at the end of the sequence, i.e. to compute P(x(t)|y(1),…,y(t)). This task is normally used when the sequence of latent variables is thought of as the underlying states that a process moves through at a sequence of points of time, with corresponding observations at each point in time. Then, it is natural to ask about the state of the process at the end. This problem can be handled efficiently using the forward algorithm (https://en.wikipedia.org/wiki/Forward_algorithm).

The forward algorithm: We derive the forward algorithm in the discrete case. As usual it can be adapted to continuous case such as the linear-Gaussian model. We use the notation $\mathbf{x}_{1:s} = (\mathbf{x}_1, \dots, \mathbf{x}_s)$. The joint distribution (of what is referred to as a hidden Markov model) takes the form

$$\mathbb{P}[\mathbf{X}_{1:t}, \mathbf{Y}_{1:t}] = \prod_{s=1}^{T} \mathbb{P}[\mathbf{X}_s|\mathbf{X}_{s-1}]\, \mathbb{P}[\mathbf{Y}_s|\mathbf{X}_s].$$

Our goal is to maximize over $\mathbf{x}_t$

$$\mathbb{P}[\mathbf{X}_t = \mathbf{x}_t|\mathbf{Y}_{1:t} = \mathbf{y}_{1:t}] = \frac{\mathbb{P}[\mathbf{X}_t = \mathbf{x}_t, \mathbf{Y}_{1:t} = \mathbf{y}_{1:t}]}{\mathbb{P}[\mathbf{Y}_{1:t} = \mathbf{y}_{1:t}]}.$$

Because the denominator does not depend on $\mathbf{x}_t$, it suffices to compute the numerator.

We give a recursion for the numerator $\alpha_t(\mathbf{x}_t)$ as a function of $\mathbf{x}_t$ that takes advantage of the conditional independence relations in the model. Observe that

$$\begin{aligned}
\alpha_t(\mathbf{x}_t) &= \mathbb{P}[\mathbf{X}_t = \mathbf{x}_t, \mathbf{Y}_{1:t} = \mathbf{y}_{1:t}] \\
&= \sum_{\mathbf{x}_{t-1} \in \mathcal{S}} \mathbb{P}[\mathbf{X}_{t-1} = \mathbf{x}_{t-1}, \mathbf{X}_t = \mathbf{x}_t, \mathbf{Y}_{1:t} = \mathbf{y}_{1:t}] \\
&= \sum_{\mathbf{x}_{t-1} \in \mathcal{S}} \mathbb{P}[\mathbf{X}_{t-1} = \mathbf{x}_{t-1}, \mathbf{Y}_{1:t-1} = \mathbf{y}_{1:t-1}] \\
&\qquad \times \mathbb{P}[\mathbf{X}_t = \mathbf{x}_t | \mathbf{X}_{t-1} = \mathbf{x}_{t-1}] \, \mathbb{P}[\mathbf{Y}_t = \mathbf{y}_t | \mathbf{X}_t = \mathbf{x}_t] \\
&= \sum_{\mathbf{x}_{t-1} \in \mathcal{S}} \alpha_{t-1}(\mathbf{x}_{t-1}) \, \mathbb{P}[\mathbf{X}_t = \mathbf{x}_t | \mathbf{X}_{t-1} = \mathbf{x}_{t-1}] \, \mathbb{P}[\mathbf{Y}_t = \mathbf{y}_t | \mathbf{X}_t = \mathbf{x}_t].
\end{aligned}$$

The two conditional probabilities on the last line are known.

Returning to the linear-Gaussian case: In the linear-Gaussian case, the joint distribution is multivariate Gaussian and the conditional densities are specified entirely by their means and covariance matrices. See [Bis (https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf), Section 13.3] for the details. Let $\boldsymbol{\mu}_t$ and $\boldsymbol{\Sigma}_t$ be the mean and covariance matrix of $\mathbf{X}_t$ conditioned on $\mathbf{Y}_{1:t}$. The recursions for these quantities are the following:

$$\begin{aligned}
\boldsymbol{\mu}_t &= F \, \boldsymbol{\mu}_{t-1} + K_t(\mathbf{Y}_t - HF\boldsymbol{\mu}_{t-1}) \\
\boldsymbol{\Sigma}_t &= (I_{d_0 \times d_0} - K_t H) P_{t-1}
\end{aligned}$$

where

$$\begin{aligned}
P_{t-1} &= F \, \boldsymbol{\Sigma}_{t-1} F^T + Q \\
K_t &= P_{t-1} H^T (H P_{t-1} H^T + R)^{-1}
\end{aligned}$$

This last matrix is known as the Kalman gain matrix. The solution above is known as Kalman filtering.

2 Location tracking

We apply Kalman filtering to location tracking. We imagine that we get noisy observations about the successive positions of an object. Think of GPS measurements for instance. We seek to get a better estimate of the location of the object using the method above. See for example this blog post (https://towardsdatascience.com/optimal-estimation-algorithms-kalman-and-particle-filters-be62dcb5e83) on location tracking.

We model the true location as a linear-Gaussian system over the 2d position $(z_{1t}, z_{2t})_t$ and velocity $(\dot{z}_{1t}, \dot{z}_{2t})_t$ sampled at Δ intervals of time. Formally, the system is

$$\mathbf{X}_t = (z_{1t}, z_{2t}, \dot{z}_{1t}, \dot{z}_{2t}), \qquad F = \begin{pmatrix} 1 & 0 & \Delta & 0 \\ 0 & 1 & 0 & \Delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

In words, the velocity is unchanged, up to Gaussian perturbation. The position changes proportionally to the velocity in the corresponding dimension.

The observations $(\tilde{z}_{1t}, \tilde{z}_{2t})_t$ are modeled as

$$\mathbf{Y}_t = (\tilde{z}_{1t}, \tilde{z}_{2t}), \qquad H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}.$$

In words, we only observe the positions, up to Gaussian noise.

3 Implementing the Kalman filter

We implement the Kalman filter as described above with known covariance matrices. We take $\Delta = 1$ for simplicity. The code is adapted from [Mur (https://github.com/probml)].

We will test Kalman filtering on a simulated path drawn from the linear-Gaussian model above. The following function creates such a path and its noisy observations.

In [1]:
```
# Julia version: 1.5.1
using LinearAlgebra, Random, Distributions, Plots
```

In [2]:
```
function lgSamplePath(ss, os, F, H, Q, R, x_1, T)

    x, y = zeros(ss,T), zeros(os,T);
    x[:,1] = x_1;
    ey = zeros(os);
    rand!(MvNormal(zeros(os),R),ey);
    y[:,1] = H*x[:,1] .+ ey;
    for t = 2:T
        ex = zeros(ss);
        rand!(MvNormal(zeros(ss),Q),ex);
        x[:,t] = F*x[:,t-1] .+ ex;
        ey = zeros(os);
        rand!(MvNormal(zeros(os),R),ey);
        y[:,t] = H*x[:,t] .+ ey;
    end
    return x, y
end
```

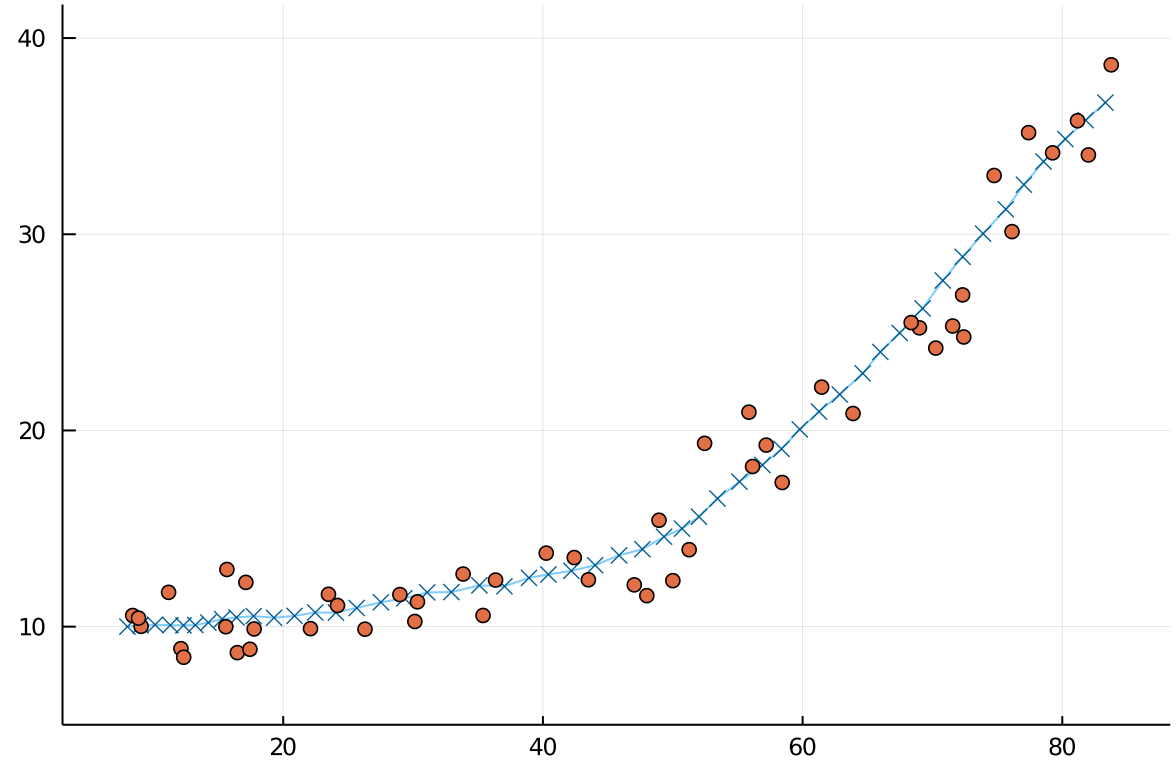
Out[2]:
```
lgSamplePath (generic function with 1 method)
```

Here is an example. Here $\mathbf{\Sigma}$ is denoted as V. In the plot, the blue crosses are the unobserved true path and the orange dots are the noisy observations.

In [3]:
```
ss = 4; # state size
os = 2; # observation size
F = [1 0 1 0; 0 1 0 1; 0 0 1 0; 0 0 0 1];
H = [1 0 0 0; 0 1 0 0];
Q = 0.01*Diagonal(ones(ss));
R = 3*Diagonal(ones(os));
initmu = [8; 10; 1; 0];
initV = 3*Diagonal(ones(ss));
T = 50;
x, y = lgSamplePath(ss, os, F, H, Q, R, initmu, T);
```

In [4]:
```
plot(x[1,:], x[2,:],
    legend=false, xlims=(minimum(x[1,:])-5,maximum(x[1,:])+5),
    ylims=(minimum(x[2,:])-5,maximum(x[2,:])+5),
    markershape=:xcross, alpha=0.5)
scatter!(y[1,:],y[2,:])
```

Out[4]:



The following function implements the Kalman filter. Here A is F and C is H. The full recursion is broken up into several steps.

In [5]:
```
function kalmanUpdate(ss, A, C, Q, R, yt, prevmu, prevV)

    mupred = A*prevmu;
    Vpred = A*prevV*A' .+ Q;
    e = yt .- C*mupred; # error (innovation)
    S = C*Vpred*C' .+ R;
    Sinv = inv(S);
    K = Vpred*C'*Sinv; # Kalman gain matrix
    munew = mupred .+ K*e;
    Vnew = (Diagonal(ones(ss)) .- K*C)*Vpred;
    return munew, Vnew
end
```

Out[5]:
```
kalmanUpdate (generic function with 1 method)
```

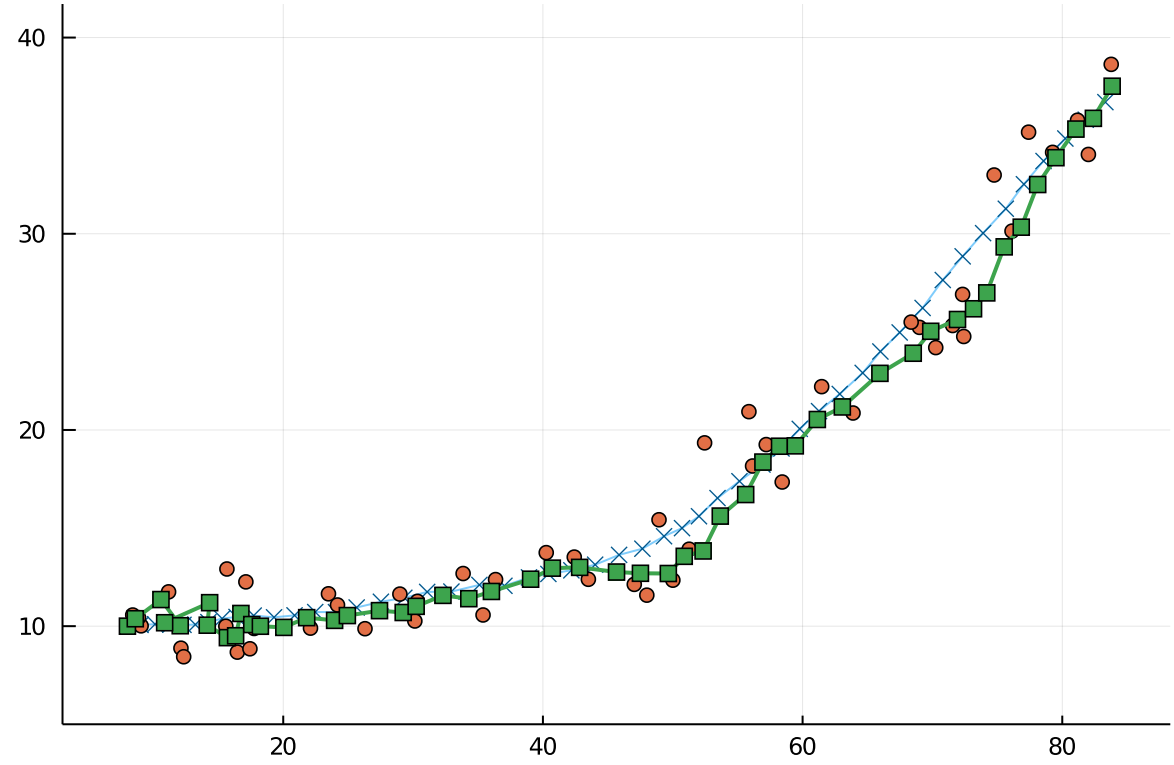
In [6]:
```
function kalmanFilter(ss, os, y, A, C, Q, R, initmu, initV, T)

    mu = zeros(ss, T);
    V = zeros(ss, ss, T);
    mu[:,1] = initmu;
    V[:,:,1] = initV;
    for t=2:T
        mu[:,t], V[:,:,t] = kalmanUpdate(ss, A, C, Q, R,
            y[:,t], mu[:,t-1], V[:,:,t-1])
    end
    return mu, V
end
```

Out[6]:
```
kalmanFilter (generic function with 1 method)
```

We apply this to the example above. The inferred unobserved states are in green.

In [7]:
```
mu, V = kalmanFilter(ss, os, y, F, H, Q, R, initmu, initV, T)
plot!(mu[1,:], mu[2,:],
    markershape=:rect, linewidth=2)
```

Out[7]:



To quantify the improvement in the inference compared to the observations, we compute the mean squared error in both cases.

In [8]:
```
dobs = x[1:2,:] - y[1:2,:]
mse_obs = sqrt(sum(dobs.^2))
```

Out[8]:
```
16.94065141843803
```

In [9]:
```
dfilt = x[1:2,:] - mu[1:2,:]
mse_filt = sqrt(sum(dfilt.^2))
```

Out[9]:
```
12.254009107771452
```

We indeed observe a reduction.

4 Missing data [optional]

We can also allow for the possibility that some observations are missing. Imagine for instance losing GPS signal while going through a tunnel. The recursions above are still valid, with the only modification that the $\mathbf{Y}_t$ and H terms are dropped at those times t where there is no observation. Julia has a convenient `missing` type to handle this situation.

We use a same sample path as above, but mask observations at times $t = 10, \ldots, 20$.

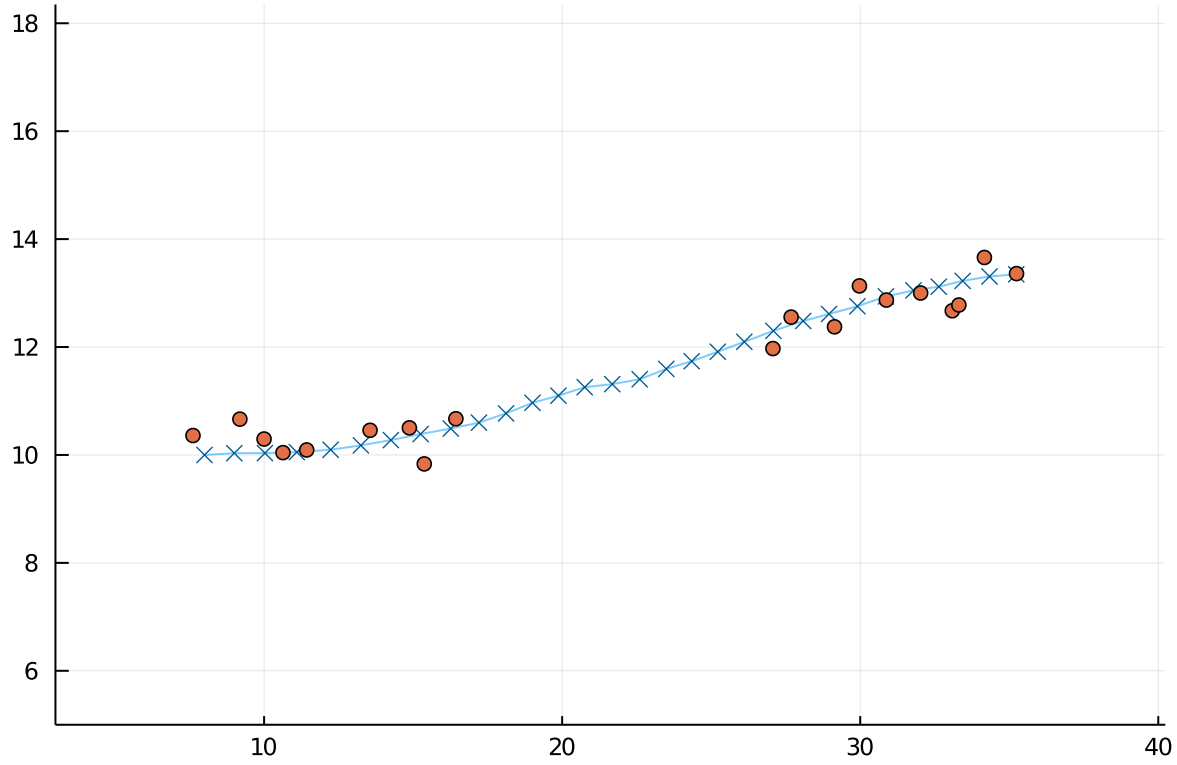
In [10]:
```
ss = 4 # state size
os = 2 # observation size
F = [1 0 1 0; 0 1 0 1; 0 0 1 0; 0 0 0 1]
H = [1 0 0 0; 0 1 0 0]
Q = 0.001*Diagonal(ones(ss))
R = 0.1*Diagonal(ones(os))
initmu = [8; 10; 1; 0]
initV = 0.1*Diagonal(ones(ss))
T = 30
x, y = lgSamplePath(ss, os, F, H, Q, R, initmu, T);
```

In [11]:
```
y = convert(Array{Union{Missing,Float64}}, y)
for i=10:20
    y[1,i]=missing
    y[2,i]=missing
end
```

Here is the sample we are aiming to infer.

In [12]:
```
plot(x[1,:], x[2,:],
    legend=false, xlims=(minimum(x[1,:])-5,maximum(x[1,:])+5),
    ylims=(minimum(x[2,:])-5,maximum(x[2,:])+5),
    markershape=:xcross, alpha=0.5)
scatter!(y[1,:],y[2,:])
```

Out[12]:



We modify the recursion accordingly.

In [13]:
```
function kalmanUpdate(ss, A, C, Q, R, yt, prevmu, prevV)

    mupred = A*prevmu
    Vpred = A*prevV*A' .+ Q
    if ismissing.(yt)==[true, true]
        return mupred, Vpred
    else
        e = yt .- C*mupred # error (innovation)
        S = C*Vpred*C' .+ R
        Sinv = inv(S)
        K = Vpred*C'*Sinv # Kalman gain matrix
        munew = mupred .+ K*e
        Vnew = (Diagonal(ones(ss)) .- K*C)*Vpred
        return munew, Vnew
    end
end
```

Out[13]:
```
kalmanUpdate (generic function with 1 method)
```

In [14]:
```
mu, V = kalmanFilter(ss, os, y, F, H, Q, R, initmu, initV, T)
plot!(mu[1,:], mu[2,:],
    markershape=:rect, linewidth=2)
```

Out[14]: