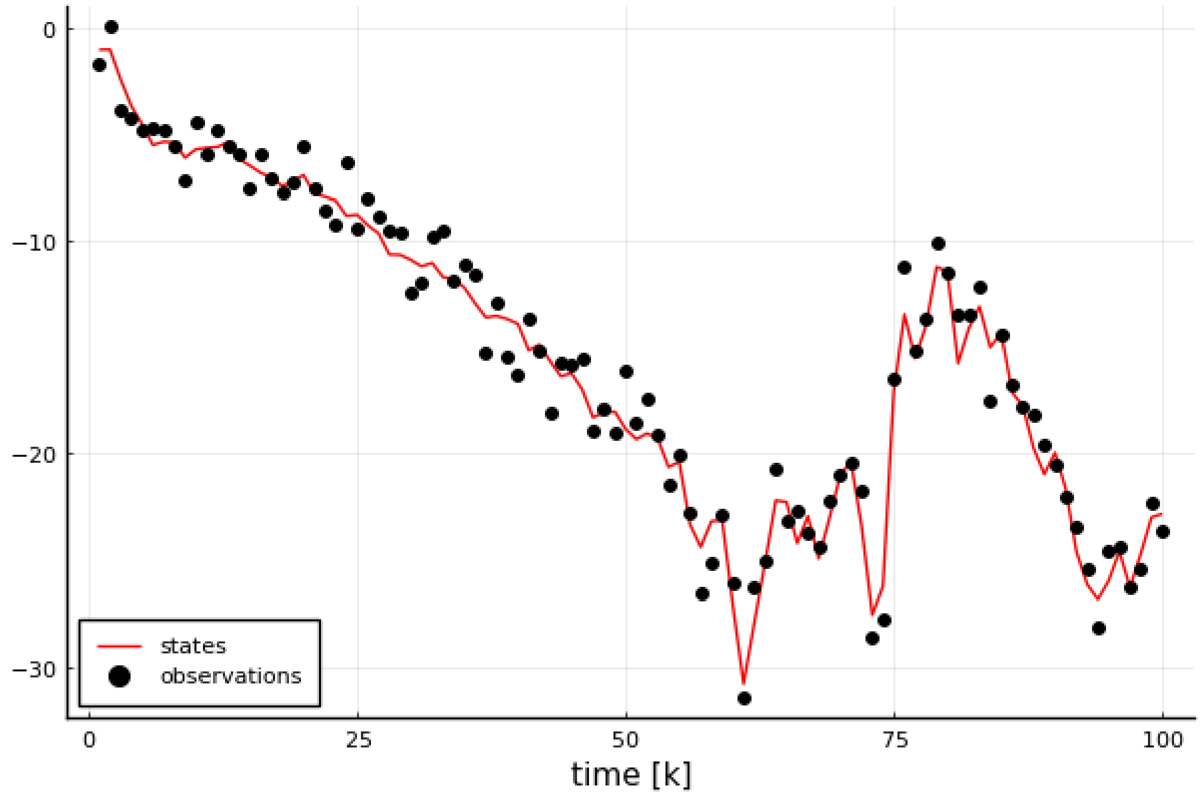
Variational Bayes for signal processing

Sioux OPT/ML Seminar

Wouter Kouw | November 5th, 2020

Suppose you're getting the following incoming signal:



Let's start modeling this signal with a standard Kalman filter. For this, I will use our toolbox FroneyLab.jl (https://github.com/biaslab/ForneyLab.jl).

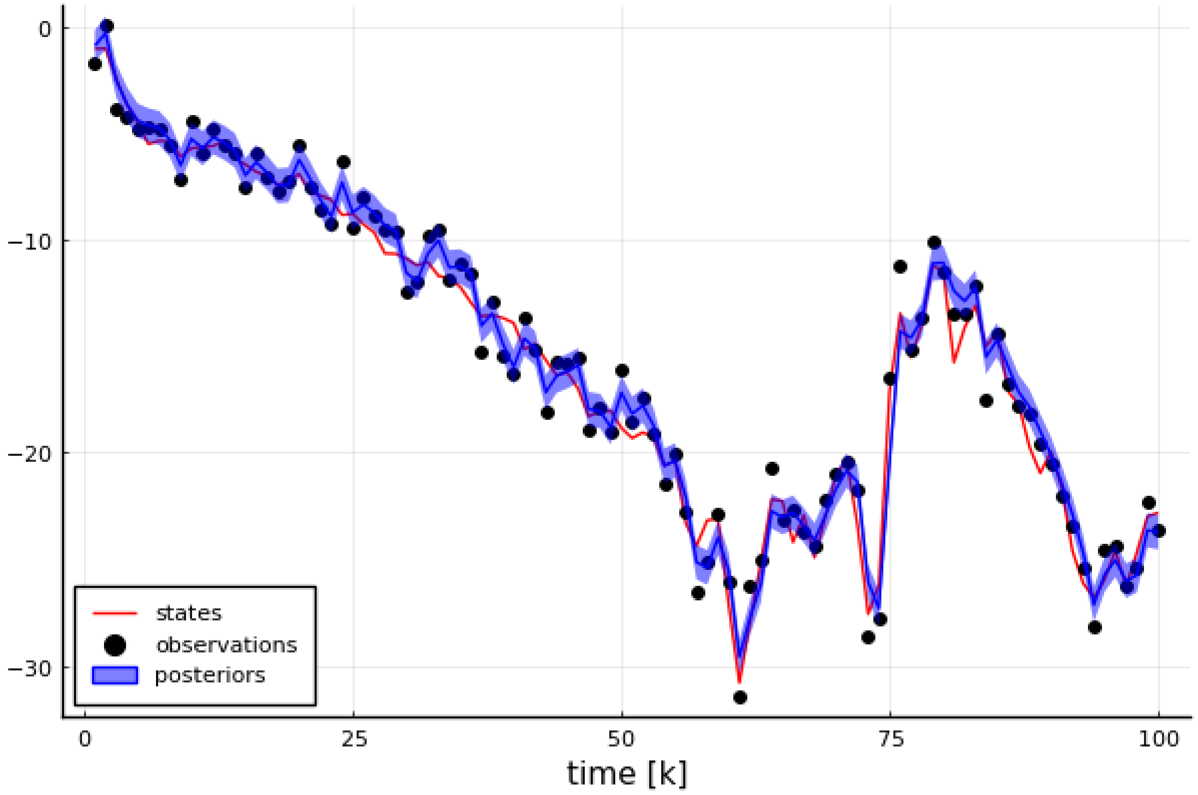
```
In [5]: # Process noise covariance matrix
        Q = [1. 0.;
             0. 1.]

        # Measurement noise variance
        R = 1.0

        # Transition matrix
        Δt = 0.1
        A = [1. Δt;
             0. 1.]

        # Emission matrix
        C = [1., 0.]

        # Previous state
        @RV x_kmin1 ~ GaussianMeanVariance(placeholder(:m_kmin1, dims=(2,)),
                                           placeholder(:S_kmin1, dims=(2,2)))

        # State transition
        @RV x_k ~ GaussianMeanVariance(A*x_kmin1, Q)

        # Observation likelihood
        @RV y_k ~ GaussianMeanVariance(dot(C, x_k), R);

        # Manually mark y_k as observed variable
        placeholder(y_k, :y_k);
```

Next, I am going to specify an inference algorithm. ForneyLab will compile one based on the model specification and we can run it to obtain estimates for each time step.

```
In [7]: # Compile algorithm and bring to scope
        algorithm = messagePassingAlgorithm(x_k)
        source_code = algorithmSourceCode(algorithm)
        eval(Meta.parse(source_code))

        # Recursive estimation
        @showprogress for k = 1:T

            # Feed data
            data = Dict(:y_k => observations[k],
                        :m_kmin1 => params_x[1][:,k],
                        :S_kmin1 => params_x[2][:,:,k])

            # Update posterior
            step!(data, posteriors)

            # Update parameter estimates
            params_x[1][:,k+1] = mean(posteriors[:x_k])
            params_x[2][:,:,k+1] = cov(posteriors[:x_k])
        end
```

```
Progress: 100%|████████████████████████████████████████| Time: 0:00:05
```

Let's see how well the model did.



So, what is happening here?

We have 1-dimensional noisy position observations y_k and 2-dimensional hidden states x_k, where the first element is the true position and the second element velocity.

I started with a state-space model of the following form:

$$\begin{aligned} x_k &= Ax_{k-1} + w_k \\ y_k &= Cx_k + v_k \end{aligned}$$

where A is a known transition matrix, $A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$, and C a known emission matrix, $C = \begin{bmatrix} 1 & 0 \end{bmatrix}$.

The process noise w_k is white: $w_k \sim \mathcal{N}(0, Q)$.

Gaussian distributions have an interesting property:

$$\text{if } z \sim \mathcal{N}(0, 1), \text{ then } x = \sigma z + \mu \implies x \sim \mathcal{N}(\mu, \sigma) .$$

We can view the state transition equation as a transformation of the noise variable:

$$x_k \sim \mathcal{N}(Ax_{k-1}, Q) .$$

Similarly, for measurement noise $v_k \sim \mathcal{N}(0, R)$, we can write:

$$y_k \sim \mathcal{N}(Cx_k, R) .$$

In this state-space model, we start with a x_0. Suppose this is also Gaussian distributed:

$$x_0 \sim \mathcal{N}(m_0, S_0)$$

Think of this as a solid guess for what the initial state is, characterized by our uncertainty surrounding it.

We want to infer the next state x_1. Using the calculus of probability, we express this as:

$$p(x_1 \mid y_1) = \frac{1}{p(y_1)} \int p(y_1 \mid x_1) p(x_1 \mid x_0) p(x_0) \mathrm{d}x_0 \ .$$

But how do you actually solve this?

We know that $p(x_1 \mid x_0) = \mathcal{N}(x_1 \mid Ax_0, Q)$ and $p(x_0) = \mathcal{N}(m_0, S_0)$. You can form a bivariate Gaussian out of these two:

$$\mathcal{N}\left(\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \mid \begin{bmatrix} m_0 \\ Am_0 \end{bmatrix}, \begin{bmatrix} S_0 & S_0 A^\top \\ AS_0 & AS_0 A^\top + Q \end{bmatrix}\right) .$$

If you then integrate out x_0, you get:

$$x_1 \sim \mathcal{N}(Am_0, AS_0 A^\top + Q) .$$

You might recognize the parameters of this distribution as the *predict* step in the classic derivation of the Kalman filter (paraphrased from Wikipedia):

$$\begin{aligned} \hat{m}_{k|k-1} &= Am_{k-1|k-1} \\ \hat{S}_{k|k-1} &= AS_{k-1|k-1} A^\top + Q . \end{aligned}$$

Since we know that $p(y_1 \mid x_1)$ is also Gaussian, we can perform the same trick to obtain a joint distribution $p(y_1, x_1)$.

Now, we are left with Bayes' rule:

$$p(x_1 \mid y_1) = \frac{p(y_1, x_1)}{p(y_1)} \; .$$

Usually, you can't compute $p(y_1)$. But with Gaussians, you can obtain a conditional distribution from a joint. For a joint distribution over x, y

$$\mathcal{N}\left(\begin{bmatrix} x \\ y \end{bmatrix} \mid \begin{bmatrix} a \\ b \end{bmatrix}, \begin{bmatrix} A & C \\ C^\top & B \end{bmatrix}\right),$$

the conditional $p(x \mid y)$ is:

$$\mathcal{N}(a + CB^{-1}(y - b), A - CB^{-1}C^\top).$$

Again, you might recognize the structure of the parameters from the *update* step for the Kalman filter:

$$m_{k|k} = \hat{m}_{k|k-1} + K_k(y_k - C\hat{m}_{k|k-1}).$$

If consider the general case, $p(x_k \mid y_{1:k})$, and kept proper track of notation, then you will find that recursively computing the posterior distribution is equivalent to Kalman filtering.

ForneyLab is designed to automatically performs all these conditioning and marginalization operations.

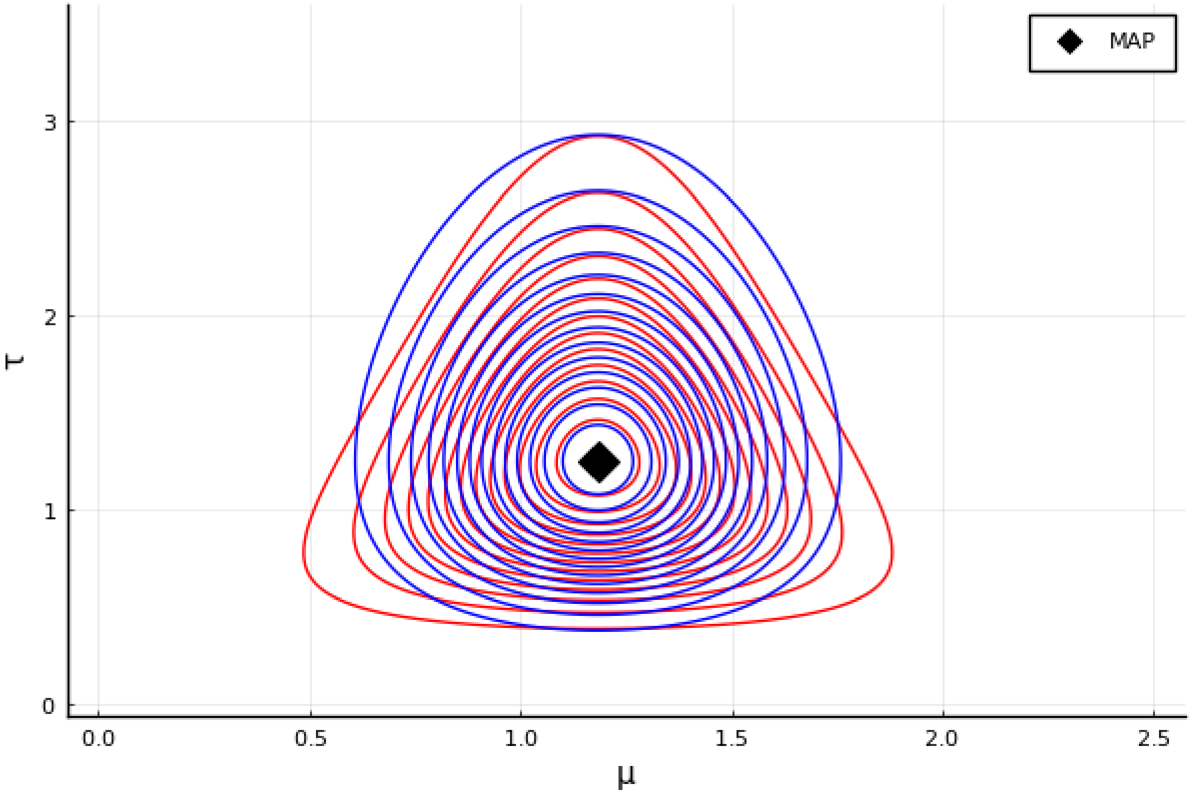
But the probabilistic perspective can do much more. For example, we can simultaneously estimate unknown (time-varying) parameters, such as process noise volatility.

Break

Not all models consist of purely Gaussian distributions. Often, we do not have analytical solutions to integrals.

In that case, we can do a form of *approximate* Bayesian inference: *Variational Bayes*.

The gist of Variational Bayes is that we approximate the intractable posterior with a tractable distribution.



Again, what is happening here?

The red contour plot you see is a *Normal-Gamma* distribution:

$$p(\mu, \tau \mid x) = \mathcal{N}\mathcal{G}(\mu, \tau \mid m, l, a, b)$$

It is a distribution of a mean parameter μ and a precision parameter τ, that come from a Gaussian likelihood function. Suppose we don't know how to perform the necessary marginalization and conditioning operations to obtain properties of this distribution.

We can pose another *simpler* distribution $q(\mu, \tau)$ that we can use to approximate the posterior $p(\mu, \tau \mid x)$ as well as possible.

But what is this $q(\mu, \tau)$?

Let the approximating, or *recognition* distribution, factorize as follows:

$$q(\mu, \tau) = q(\mu)\, q(\tau)$$

where

$$\begin{aligned} q(\mu) &\sim \mathcal{N}(m_\mu, v_\mu) \\ q(\tau) &\sim \Gamma(a_\tau, b_\tau)\,. \end{aligned}$$

In words: we are going to approximate the posterior distribution with the product of a Normal distribution and a Gamma distribution, that are assumed to be independent of each other.

We measure how well q approximates p with an objective function known as the Free Energy function:

$$\mathcal{F}[q] = \int q(\mu)q(\tau) \log \frac{q(\mu)q(\tau)}{p(\mu, \tau, x)} \mathrm{d}\mu \mathrm{d}\tau \ ,$$

Note that this function contains the joint $p(\mu, \tau, x)$ instead of the posterior. Remember that the joint can be written as: $p(\mu, \tau \mid x)p(x)$. That is the posterior times "model evidence".

So, the Free Energy function contains both the Kullback-Leibler divergence between the true posterior p and the approximing distribution q, as well as the model evidence which allow you to compare models.

But what are we optimizing over?

We want to change $q(\mu)$ and $q(\tau)$ to improve the approximation. You change distributions by changing parameters. So, the answer is that we are optimizing over the parameters of the two q's.

You can derive the analytic form of the optimal recognition factors:

$$q^*(\mu) \propto \exp\left(\mathbb{E}_{q(\tau)}[-\log \mathcal{N}(x|\mu,\tau)] + \mathbb{E}_{q(\tau)}[-\log \mathcal{N}\mathcal{G}(\mu,\tau|m_0,l_0,a_0,b_0)]\right)$$
$$q^*(\tau) \propto \exp\left(\mathbb{E}_{q(\mu)}[-\log \mathcal{N}(x|\mu,\tau)] + \mathbb{E}_{q(\mu)}[-\log \mathcal{N}\mathcal{G}(\mu,\tau|m_0,l_0,a_0,b_0)]\right)$$

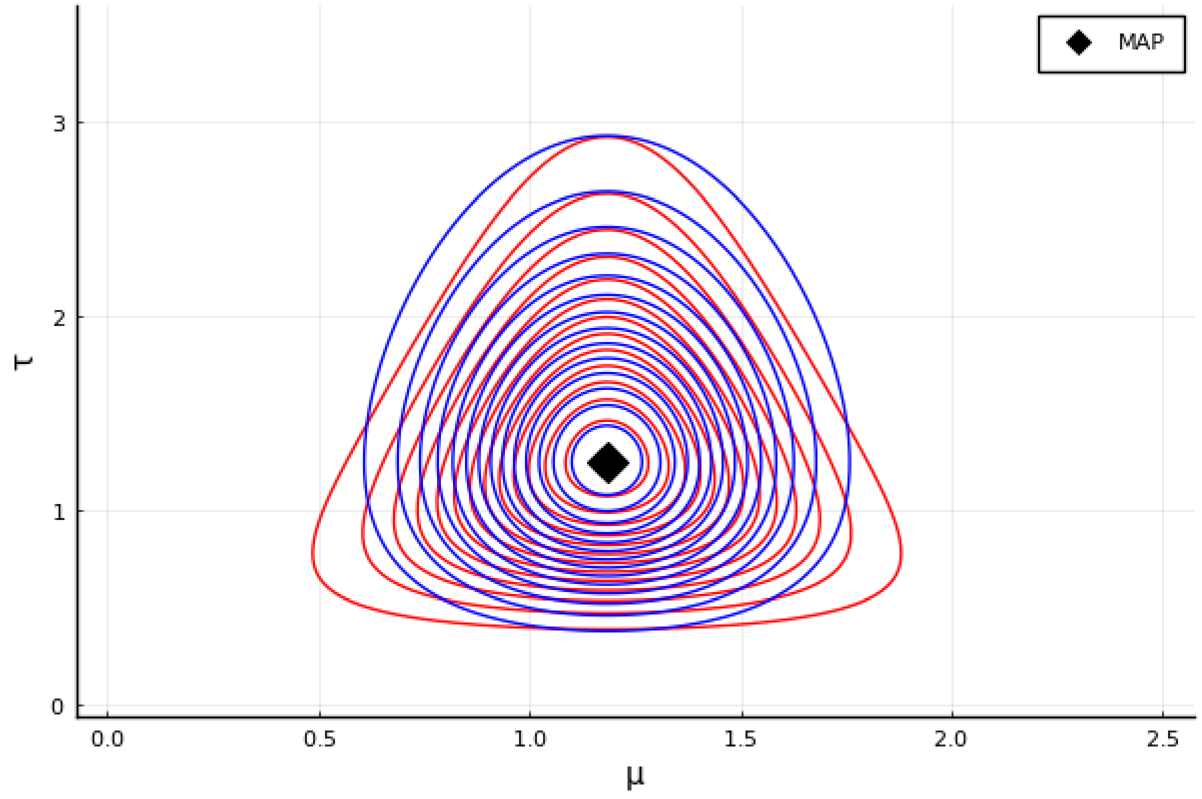
If you work out all the expectations, then you get analytic expressions for the parameters of the recognition distributions:

$$\begin{aligned}
m_\mu^* &= (n\bar{x} + l_0 m_0) \,/\, (n + l_0) \\
v_\mu^* &= \left(\frac{a_\tau}{b_\tau}(n + l_0)\right)^{-1} \\
a_\tau^* &= a_0 + \frac{n+1}{2} \\
b_\tau^* &= b_0 + \frac{1}{2}\left[l_0(m_0^2 - 2m_0 m_\mu + m_\mu^2 + v_\mu) + n(\tilde{x} - 2\bar{x}m_\mu + m_\mu^2 + v_\mu)\right] .
\end{aligned}$$

Note that each parameter update depends on the other parameters. We therefore initialize them and iteratively update them.

==> This is a coordinate descent algorithm.

If we look back at the animation, you can see that we iteratively update the parameters for $q(\mu)$ and $q(\tau)$:



Break

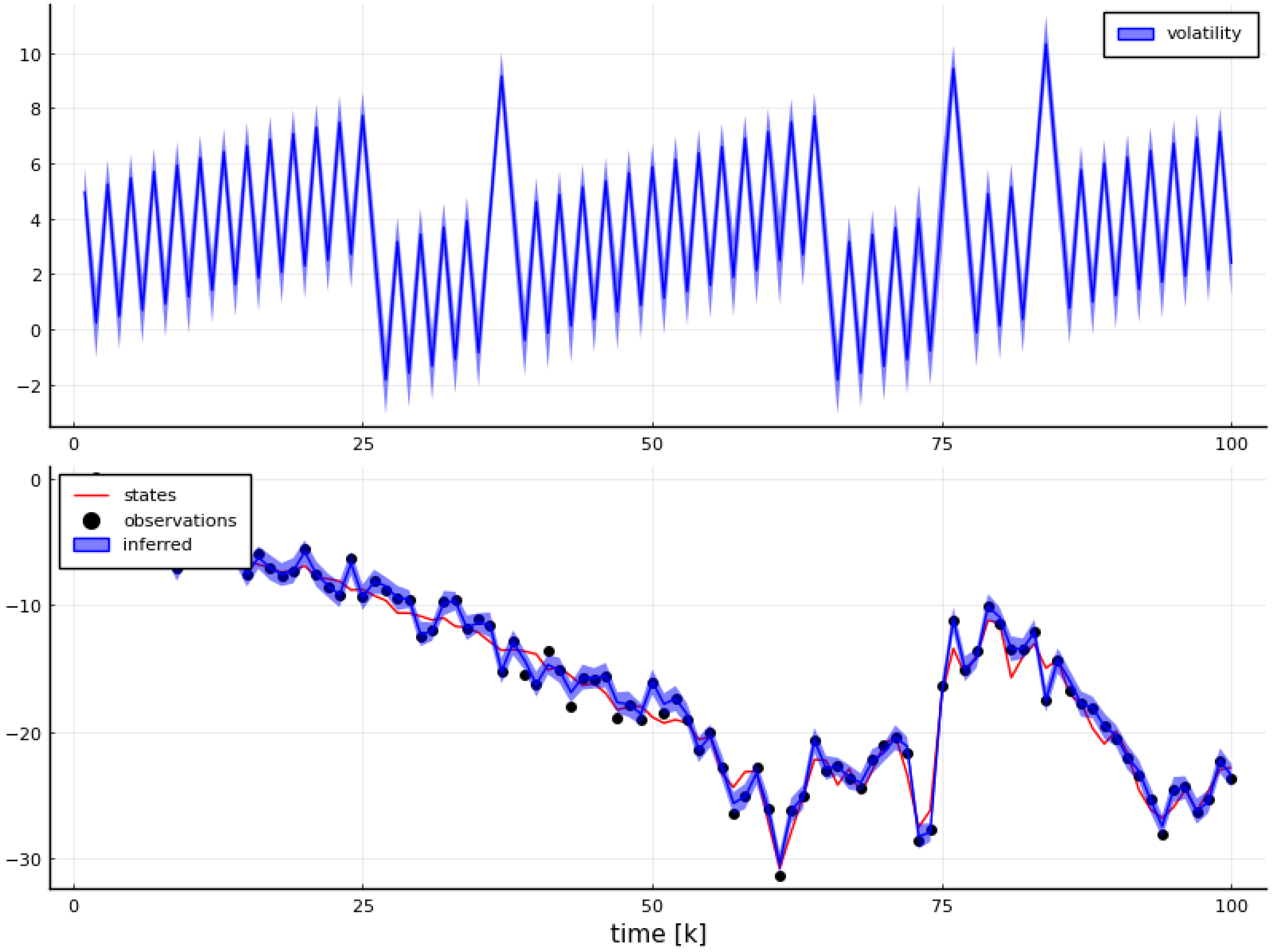
At BIASlab, we use the Free Energy Minimisation framework to design more complicated signal processing systems.

We work a lot on hierarchical models that induce time-varying parameter estimates. One example of this is our Hierarchical Gaussian Filter. It is essentially a Kalman filter, with an higher-layer random walk that governs process noise covariance.

In [12]:

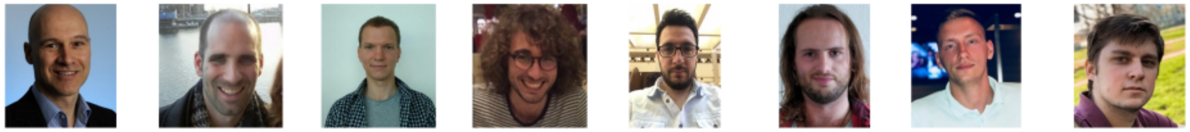
```
# Import custom GaussianControlledVariance node
using GCV

# Start graph
graph = FactorGraph();

# Volatility parameters
κ = 0.9
ω = 0.0
γ = 0.005

# Previous volatility
@RV z_kmin1 ~ GaussianMeanVariance(placeholder(:m_z_kmin1), placeholder(:v_z_kmi
n1))

# Current volatility
@RV z_k ~ GaussianMeanPrecision(z_kmin1, γ)

# Previous state
@RV x_kmin1 ~ GaussianMeanVariance(placeholder(:m_x_kmin1), placeholder(:v_x_kmi
n1))

# Current state
@RV x_k ~ GaussianControlledVariance(x_kmin1, z_k, κ, ω)

# Current observation
@RV y_k ~ GaussianMeanVariance(x_k, R)

# Data placeholder
placeholder(y_k, :y_k);
```

```
In [13]: q = PosteriorFactorization([z_kmin1; z_k],[x_kmin1; x_k]; ids=[:Z,:X])
         algorithm = messagePassingAlgorithm(free_energy=true)
         source_code = algorithmSourceCode(algorithm, free_energy=true);
         eval(Meta.parse(source_code))

         # Recursive estimation
         @showprogress for k = 1:T

             # Feed data
             data = Dict(:y_k => observations[k],
                         :m_x_kmin1 => params_x[1][k],
                         :v_x_kmin1 => params_x[2][k],
                         :m_z_kmin1 => params_z[1][k],
                         :v_z_kmin1 => params_z[2][k])

             # Iteratively update recognition factors
             for i = 1:10
                 stepX!(data, posteriors)
                 stepZ!(data, posteriors)
             end

             # Update parameter estimates
             params_x[1][k+1] = mean(posteriors[:x_k])
             params_x[2][k+1] = cov(posteriors[:x_k])
             params_z[1][k+1] = mean(posteriors[:z_k])
             params_z[2][k+1] = cov(posteriors[:z_k])
         end

         Progress: 100%|██████████████████████████████████████████| Time: 0:00:20
```


volatility
states
observations
inferred
time [k]
10
8
6
4
2
0
−2
0
−10
−20
−30
0
25
50
75
100

Questions?

BIASlab:


Bert de Vries   Thijs van de Laar   Ivan Bocharov   Ismail Senoz   Semih Akbayrak   Magnus Koudahl   Albert Podusenko   Dmitry Bagaev

References:

- [ForneyLab.jl (https://github.com/biaslab/ForneyLab.jl)](https://github.com/biaslab/ForneyLab.jl)
- [Hierarchical Gaussian Filter (https://biaslab.github.io/pdf/mlsp2018/senoz_mlsp_2018.pdf)](https://biaslab.github.io/pdf/mlsp2018/senoz_mlsp_2018.pdf)
- [Simon Särkkä - Bayesian Filtering & Smoothing (https://www.cambridge.org/core/books/bayesian-filtering-and-smoothing/C372FB31C5D9A100F8476C1B23721A67)](https://www.cambridge.org/core/books/bayesian-filtering-and-smoothing/C372FB31C5D9A100F8476C1B23721A67)