# ForneyLab.jl: a Julia Toolbox for Factor Graph-based Probabilistic Programming *

Thijs van de Laar, Marco Cox and Bert de Vries

Eindhoven University of Technology, Eindhoven, NL
Email: t.w.v.d.laar@tue.nl

**Abstract**

Scientific modeling concerns a continual search for better models for given data sets. This process can be elegantly captured in a Bayesian inference framework. ForneyLab enables largely automated scientific design loops by deriving fast, analytic algorithms for approximate Bayesian inference.

## ForneyLab

Scientific modeling concerns a continual search for better models for given data sets. This process can be elegantly captured in a Bayesian (probabilistic) inference framework. ForneyLab is an open source probabilistic programming toolbox that enables largely automated design loops by deriving fast, analytic algorithms for message passing-based approximate Bayesian inference.

Probabilistic programming extends conventional programming languages with the facility to compute rationally with random variables through probability calculus. ForneyLab is a Julia package that allows the user to specify a probabilistic model and pose inference problems on those models. In return, FL automatically constructs a Julia program that executes a message passing-based approximate inference procedure. In conjunction with a specification of (possibly streaming) data sources, the inference program can be executed to fit the data to the model. Additionally, ForneyLab provides a measure of the performance of the data fit, thus facilitating comparison to alternative models. Typical applications include the design of dynamic models for (Bayesian) machine learning systems, statistical signal processing, computational neuroscience, etc.

More specifically, a design cycle in FL consists of three phases. First, in the *build phase*, the end user specifies a (probabilistic) model. Through the use of macros, the model is specified in a domain-specific syntax that strongly resembles notational conventions in alternative probabilistic programming languages.

*Presented at JuliaCon 2018. Presentation available at https://youtu.be/RS4hJ4oBr9c .

Usually, even complex model specifications fit on less than one page of code. Under the hood, ForneyLab builds a *Forney-style factor graph* (FFG), which is a computational network representation of the model. A strong feature of the FFG formalism includes its extremely modular make-up, which allows re-use of computational inference primitives. The choice for FFG-based model specifications (and consequently, message passing-based inference procedures) is where ForneyLab differs from competing probabilistic programming languages (aside from our choice for a native Julia realization).

Next, in the *schedule phase*, the user specifies the inference problem. This is usually encoded by a few lines of code. ForneyLab then automatically derives a message passing algorithm that, when executed, computes the posterior marginal probability over the desired variables. The generated message passing code may contain thousands of code lines, depending on the size of the model. A clear asset of message passing-based inference algorithms is that they are comprised of many cheap and analytical updates that can be re-used across models, and furthermore can potentially be implemented on dedicated (parallel) hardware configurations. This contrasts to modern sampling-based approaches to inference, such as Markov chain Monte Carlo and Automatic Differentiation Variational Inference, which usually require massive computational resources.

In the final *infer phase*, ForneyLab parses and executes the automatically generated inference program. For the user, this action is initiated by one statement. Additionally, a separate function can be generated to evaluate the model fit, which provides insights on model quality and algorithm convergence during inference.

ForneyLab relies heavily on Julia's meta-programming functionality. Not only does it use macros for the model specification, but the main output is also a Julia program by itself. This flexibility, together with benefits of the modular FFG approach, makes it a powerful tool for a scientist or engineer who wants to develop models for a given data set.

# Example

Assume that we have time-dependent noisy observations `y_t` of the position of a (one-dimensionally moving) car, and that we are interested in inferring (tracking) the hidden trajectory `x_t` of the car in real-time.

First, we *build* a probabilistic model, which expresses our belief for how the data `y_t` are generated from the hidden state `x_t`:

```
# placeholders for prior statistics
x_t_prior_m = placeholder(:x_t_prior_m)
x_t_prior_v = placeholder(:x_t_prior_v)

@RV x_t_prev ~ GaussianMeanVariance(x_t_prior_m, x_t_prior_v)
    # state prior
@RV x_t ~ GaussianMeanVariance(x_t_prev, 0.05) # state transition
@RV y_t ~ GaussianMeanVariance(x_t, 2.0) # observation

placeholder(y_t, :y_t) # placeholder for data
```

Next, we *schedule* the algorithm by stating our inference problem (tracking the hidden state `x_t`):

```
algo = sumProductAlgorithm(x_t)
```

The resulting algorithm is a Julia function that accepts a data dictionary (shown below). The algorithm computes consecutive messages through tabulated (analytic) update rules. These messages are used as intermediate results, (e.g., the computation for message two requires message one), and in the end combine to the marginal posterior result (the marginal for `x_t`). Explicitly cleaning out the intermediate results of these message sequences would reveal that the algorithm evaluates to the well-known Kalman filtering result. However, a big advantage of the followed procedure is that this result is derived automatically by ForneyLab without burdening the end user with algebraic details. The following code fragment is automatically generated:

```
function step!(data::Dict, marginals::Dict=Dict(),
    messages::Vector{Message}=Array{Message}(3))

    messages[1] = ruleSPGaussianMeanVarianceOutVPP(nothing,
      Message(Univariate, PointMass, m=data[:x_t_prior_m]),
      Message(Univariate, PointMass, m=data[:x_t_prior_v]))
    messages[2] = ruleSPGaussianMeanVarianceOutVGP(nothing,
      messages[1], Message(Univariate, PointMass, m=0.05))
    messages[3] = ruleSPGaussianMeanVarianceMPVP(
      Message(Univariate, PointMass, m=data[:y_t]), nothing,
      Message(Univariate, PointMass, m=2.0))

    marginals[:x_t] = messages[2].dist * messages[3].dist

    return marginals
end
```

Finally, we *infer* the result by recursively applying this algorithm to an online data stream. After parsing and evaluating the generated program, a single time-step is evaluated as

```
eval(parse(algo))

data = Dict(:y_t         => y_t, # new datum
            :x_t_prior_m => x_t_prior_m, # prior statistics
            :x_t_prior_v => x_t_prior_v)

marginals = step!(data) # infer posterior

x_t_post_m = mean(marginals[:x_t])
x_t_post_v = var(marginals[:x_t])
```

This demo illustrates how ForneyLab uses Julia, and how the toolbox automatically and flexibly derives complex algorithms through an intuitive user interface.