International Journal of Approximate Reasoning 104 (2019) 185–204

Contents lists available at ScienceDirect

International Journal of Approximate Reasoning

www.elsevier.com/locate/ijar

A factor graph approach to automated design of Bayesian signal processing algorithms

Marco Cox [a,*,1], Thijs van de Laar [a,*,1], Bert de Vries [a,b]

[a] Department of Electrical Engineering, Eindhoven University of Technology, PO Box 513, 5600 MB, Eindhoven, the Netherlands
[b] GN Hearing, Het Eeuwsel 6, 5612 AS, Eindhoven, the Netherlands

ARTICLE INFO

Article history:
Received 27 June 2018
Received in revised form 23 October 2018
Accepted 1 November 2018
Available online 8 November 2018

Keywords:
Probabilistic programming
Bayesian inference
Julia
Factor graphs
Message passing

ABSTRACT

The benefits of automating design cycles for Bayesian inference-based algorithms are becoming increasingly recognized by the machine learning community. As a result, interest in probabilistic programming frameworks has much increased over the past few years. This paper explores a specific probabilistic programming paradigm, namely message passing in Forney-style factor graphs (FFGs), in the context of automated design of efficient Bayesian signal processing algorithms. To this end, we developed "ForneyLab"[2] as a Julia toolbox for message passing-based inference in FFGs. We show by example how ForneyLab enables automatic derivation of Bayesian signal processing algorithms, including algorithms for parameter estimation and model comparison. Crucially, due to the modular makeup of the FFG framework, both the model specification and inference methods are readily extensible in ForneyLab. In order to test this framework, we compared variational message passing as implemented by ForneyLab with automatic differentiation variational inference (ADVI) and Monte Carlo methods as implemented by state-of-the-art tools "Edward" and "Stan". In terms of performance, extensibility and stability issues, ForneyLab appears to enjoy an edge relative to its competitors for automated inference in state-space models.

1. Introduction

The design of signal processing algorithms by probabilistic modeling comprises an iterative process that involves three phases: (1) model specification, (2) probabilistic inference (i.e., the actual algorithm derivation) and (3) performance evaluation (i.e., scoring of the algorithm). In this framework, a (signal processing) algorithm is defined as an inference task on a probabilistic model. For example, a Kalman filter-based algorithm can be specified as an inference task on a linear Gaussian dynamical system.

Based on the algorithm scoring results (phase 3), one might revise the model specification and repeat the process. In [1], this "build, compute, critique, repeat"-cycle is called "Box's loop" and a strong argument can be made that this iterative process realizes the general scientific method. The great promise of a probabilistic modeling approach to algorithm design is that both the inference and scoring phases (phases 2 and 3) are results of Bayesian inference and therefore in principle

[*] Corresponding authors.
E-mail addresses: m.g.h.cox@tue.nl (M. Cox), t.w.v.d.laar@tue.nl (T. van de Laar).

[1] Joint first authors, order decided by coin toss.

[2] ForneyLab is available for download at https://github.com/biaslab/ForneyLab.jl.

https://doi.org/10.1016/j.ijar.2018.11.002
0888-613X/© 2018 Elsevier Inc. All rights reserved.

automatable. If indeed phases 2 and 3 were automated by a suitable software suite, then a (human) algorithm designer could quickly loop through design iterations by proposing alternative models until a satisfactory performance score has been reached.

In practice, fully automating "inference" and "scoring" phases by a Bayesian inference toolbox is a yet unsolved problem. This has a limiting effect on the number of affordable iterations through Box's loop, which ultimately limits the quality of the final result.

In order to reduce the time and effort spent in the "inference" and "scoring" phases, an extensive line of research has focused on automating the derivation and implementation of (Bayesian) inference algorithms, dating back (at least) to the BUGS project that started in 1989 [2]. The general idea behind this *probabilistic programming* approach is to develop software that accepts a probabilistic model specification and returns an (approximate) Bayesian inference algorithm, without the need for manual derivations. Historically, probabilistic programming systems have relied heavily on Markov chain Monte Carlo (MCMC) methods due to their broad applicability. More recently, techniques like black-box variational inference (BBVI) [3] have been added to the mix, for example in the popular probabilistic programming packages Stan [4] and Edward [5].

Automatic generation of probabilistic inference algorithms involves an important trade-off between generality and efficiency. Inference methods that can automatically be applied to a wide array of models are usually not the most efficient, due to their black-box nature that prevents exploitation of model-specific properties. For example, MCMC methods are very generic, but can be orders of magnitude slower than inference algorithms that exploit model-specific properties such as conjugacy. This makes MCMC-based methods less suitable for situations that require real-time data processing or setups with limited computational resources. On the other hand, computationally more efficient methods such as exact Bayesian inference, variational Bayesian inference and expectation propagation require model-specific derivations, which makes it harder to generate them automatically for arbitrary models. Recent work has focused on the design of Bayesian inference algorithms that are more efficient than vanilla MCMC methods while still being broadly applicable [3,6,7].

In this paper we focus on message passing in factor graphs as a platform for automated design of Bayesian inference algorithms. Message passing exploits local model structure, while retaining general applicability. For instance, inference algorithms such as belief propagation [8], variational Bayes [9], expectation propagation [10] and particle filtering [11] have already been formulated as message passing algorithms. The appeal of message passing is mainly due to its divide-and-conquer approach to inference, which allows it to marry the computational efficiency of analytic methods with the generality of black-box methods.

The goal of this paper is to paint a spectrum of possibilities that arise when adhering to the message passing approach to Bayesian inference, with a focus on time series modeling. Throughout the paper we present concrete examples that are implemented with ForneyLab, a novel publicly available toolbox we developed for generating message passing algorithms on Forney-style factor graphs [12].

After a short technical introduction (Sec. 2), we illustrate how the message passing approach to inference with ForneyLab enables

- automated design of message passing algorithms (Sec. 3);
- effective and flexible model design (Sec. 4);
- efficient Bayesian inference (Sec. 5).

Finally, we discuss related work (Sec. 6), and conclude (Sec. 7) by connecting ideas from the literature with the present framework.

2. Background: Forney-style factor graphs and message passing algorithms

This section provides a short technical summary of message passing-based inference on Forney-style factor graphs. A more extensive introduction is available in [13]. Furthermore, a detailed description of message passing on Forney-style factor graphs in the context of signal processing is available in [14].

2.1. Message passing on Forney-style factor graphs

A Forney-style factor graph (FFG) [12] offers a graphical representation of a factorized probabilistic model. In an FFG, edges represent variables and nodes specify relations between variables. As a simple example, consider a generative model (joint probability distribution) over variables $x_1, \dots, x_5$ that factors as

$$f(x_1, \dots, x_5) = f_a(x_1)\, f_b(x_1, x_2)\, f_c(x_2, x_3, x_4)\, f_d(x_4, x_5)\,, \tag{1}$$

where $f_\bullet(\cdot)$ denotes a probability density function. This factorized model can be represented graphically as an FFG, as shown in Fig. 1. Note that although an FFG is principally an undirected graph, in the case of generative models we specify a direction for the edges to indicate the "generative direction". The edge direction simply anchors the direction of messages flowing on the graph (we speak of forward and backward messages that flow with or against the edge direction, respectively). In other words, the edge directionality is purely a notational issue and has no computational consequences.

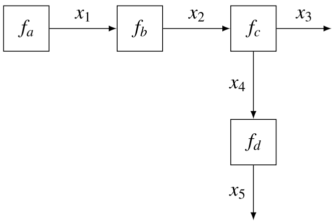

Fig. 1. Forney-style factor graph (FFG) representation of Eq. (1). In an FFG, edges correspond to variables and nodes represent factors that encode constraints among variables. A node connects to all edges that correspond to variables that occur in its factor function. For example, node f_b connects to edges x_1 and x_2 since those variables occur in $f_b(x_1, x_2)$. Variables that occur in just one factor (x_3 and x_5 in this case) are represented by half-edges. While an FFG is principally an undirected graph, we usually specify a direction for the (half-)edges to indicate the generative direction of the model and to anchor the direction of messages flowing on the graph.

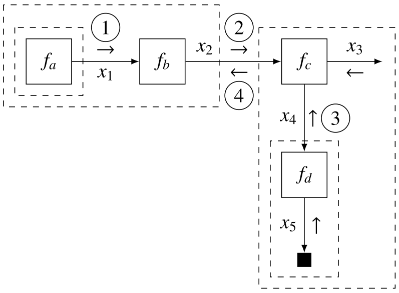

Fig. 2. Visualization of the message passing schedule corresponding to Eq. (2b) with observed variable $x_5 = \hat{x}_5$. The observation is indicated by terminating edge x_5 by a small solid node that technically represents the factor $\delta(x_5 - \hat{x}_5)$. Messages are represented by numbered arrows, and the message sequence is chosen such that there are only backward dependencies. Dashed boxes mark the parts of the graph that are covered by the respective messages coming out of those boxes. The marginal posterior distribution $f(x_2 \mid x_5 = \hat{x}_5)$ is obtained by taking the product of the messages that flow on edge x_2 and normalizing.

The FFG representation of a probabilistic model helps to automate probabilistic inference tasks. As an example, consider we observe $x_5 = \hat{x}_5$ and are interested in calculating the marginal posterior probability distribution of x_2 given this observation.

In the FFG context, observing the realization of a variable leads to the introduction of an extra factor in the model which "clamps" the variable to its observed value. In our example where x_5 is observed at value $\hat{x}_5$, we extend the generative model to $f(x_1, \dots, x_5) \cdot \delta(x_5 - \hat{x}_5)$. Following the notation introduced in [15], we denote such "clamping" factors in the FFG by solid black nodes. The FFG of the extended model is illustrated in Fig. 2.

Computing the marginal posterior distribution of x_2 under the observation $x_5 = \hat{x}_5$ involves integrating the extended model over all variables except x_2, and renormalizing:

$$f(x_2 \mid x_5 = \hat{x}_5) \propto \int \cdots \int f(x_1, \dots, x_5) \cdot \delta(x_5 - \hat{x}_5) \, \mathrm{d}x_1 \, \mathrm{d}x_3 \, \mathrm{d}x_4 \, \mathrm{d}x_5 \tag{2a}$$

$$= \overbrace{\int \underbrace{f_a(x_1)}_{\textcircled{1}} \, f_b(x_1, x_2) \, \mathrm{d}x_1}^{\textcircled{2}} \; \overbrace{\iint f_c(x_2, x_3, x_4) \underbrace{\left(\int f_d(x_4, x_5) \cdot \delta(x_5 - \hat{x}_5) \, \mathrm{d}x_5 \right)}_{\textcircled{3}} \mathrm{d}x_3 \, \mathrm{d}x_4}^{\textcircled{4}} \, . \tag{2b}$$

The nested integrals in Eq. (2b) result from substituting the factorization of Eq. (1) and rearranging the integrals according to the distributive law. Rearranging large integrals of this type as a product of nested sub-integrals can be automated by exploiting the FFG representation of the corresponding model. The sub-integrals indicated by circled numbers correspond to integrals over parts of the model (indicated by dashed boxes in Fig. 2), and their solutions can be interpreted as messages flowing on the FFG. Therefore, this procedure is known as *message passing* (or summary propagation). The messages are ordered ("scheduled") in such a way that there are only backward dependencies, i.e., each message can be calculated from preceding messages in the schedule. Crucially, these schedules can be generated automatically, for example by performing a depth-first search on the FFG.

Message passing is generally efficient because the computation of every message is node-local in the FFG. More specifically, the message flowing out of a factor node f_a can be calculated from the analytic form of factor f_a and all messages inbound to node f_a. If the analytic forms of the incoming messages are known (which is often the case), a pre-derived *message computation rule* can be used to compute the outgoing message. These rules can be stored in a lookup table for

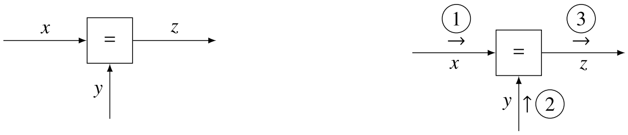

Fig. 3. FFG representation (left) and message passing schedule (right) for an equality constraint node.

reuse in any model that involves that specific factor-message combination. This important locality property thus enables efficient and automated probabilistic inference.

In the case of marginalization, the messages are derived according to the so-called *sum-product rule*,[3] which leads to the sum-product (belief propagation) algorithm. As an example derivation we consider the outgoing message of an "equality constraint node" (see Fig. 3, left; see also [14]), which constrains three variables x, y, z to equal values through the factor $f_=(x, y, z) = \delta(z - x)\,\delta(z - y)$.

For given incoming messages $\mu_1(x)$ and $\mu_2(y)$ on edges x and y (depicted by ① and ② in Fig. 3, right), the outgoing sum-product message on the z-edge is given by

$$\mu_3(z) = \iint \mu_1(x)\,\mu_2(y)\,f_=(x, y, z)\,\mathrm{d}x\,\mathrm{d}y \tag{3a}$$

$$= \mu_1(z)\,\mu_2(z)\,. \tag{3b}$$

Note that the outgoing message is only a function of z and that it is calculated from only node-local information, namely the incoming messages at the corresponding node and the definition of the node factor itself. Equality constraint nodes are quite prevalent in FFGs because they constitute a branching mechanism that distributes variables over multiple (more than two) factors in the graph. If we interpret message $\mu_1(\cdot)$ as a prior and message $\mu_2(\cdot)$ as a likelihood function, then message $\mu_3(\cdot)$ becomes proportional to the posterior distribution over z. Therefore, message passing through the equality node effectively fuses information from two sources by executing Bayes rule (up to a normalizing constant).

2.2. Variational message passing

Inference problems on practical models often lead to sub-integrals that are difficult to evaluate analytically. In such cases, exact inference through sum-product message passing is impractical. However, one can often resort to alternative message passing algorithms that yield approximate solutions. One such algorithm is variational message passing (VMP) [9,16], which will be applied extensively throughout this paper.

VMP is the message passing implementation of variational Bayesian inference, which finds an approximate solution to an inference problem by reformulating inference as an optimization task [17]. Concretely, for a given model $p(\boldsymbol{y}, \boldsymbol{z})$ with hidden variables $\boldsymbol{z}$ and observed variables $\boldsymbol{y}$, we define an approximate inference solution $q(\boldsymbol{z}) \approx p(\boldsymbol{z}\,|\,\boldsymbol{y})$ (also known as the *recognition* distribution), and a so-called variational free energy functional

$$F[q] \triangleq \underbrace{-\int_{\boldsymbol{z}} q(\boldsymbol{z}) \log p(\boldsymbol{y}, \boldsymbol{z})\,\mathrm{d}\boldsymbol{z}}_{\text{energy}} + \underbrace{\int_{\boldsymbol{z}} q(\boldsymbol{z}) \log q(\boldsymbol{z})\,\mathrm{d}\boldsymbol{z}}_{-\text{entropy}} \tag{4a}$$

$$= \underbrace{-\log p(\boldsymbol{y})}_{-\text{log-evidence}} + \underbrace{\int_{\boldsymbol{z}} q(\boldsymbol{z}) \log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z}|\boldsymbol{y})}\,\mathrm{d}\boldsymbol{z}}_{\text{KL-divergence}}\,. \tag{4b}$$

In the machine learning community, the *negative* free energy ($-F[q]$) is also known as the Evidence Lower BOund (ELBO) [18]. The variational Bayes algorithm proceeds by minimizing $F[q]$ with respect to the parameters of the approximate posterior $q(\boldsymbol{z})$ through some optimization procedure. Since only the second term in Eq. (4b) involves $q(\boldsymbol{z})$, this is equivalent to minimizing the Kullback–Leibler (KL) divergence between the proposed solution $q(\boldsymbol{z})$ and the (perfect) Bayesian solution $p(\boldsymbol{z}|\boldsymbol{y})$. As a result, the approximate solution $q(\boldsymbol{z})$ is optimized to be as close as possible to the exact solution $p(\boldsymbol{z}|\boldsymbol{y})$ in terms of KL-divergence. Note also that if the minimized KL-divergence is small in comparison to the first term (minus-log-evidence), then the optimized free energy is a good approximation of the Bayesian model evidence. In practice, $q^* = \arg\min F[q]$ is used as a proxy for the target posterior, and $F[q^*]$ is often used to score the model performance. The analytic form of the approximate posterior $q(\boldsymbol{z})$ is usually chosen such that it can provide a reasonable approximation to the true posterior while keeping the optimization problem tractable.

[3] The name sum-product rule derives from the observation that each sub-integral in Eq. (2b) comprises a sum (integral) of a product of factors.

In [16], it is shown that variational free energy minimization as outlined above can be implemented by message passing on the FFG representation of the generative model. Similar to sum-product message passing, VMP involves the evaluation of a sequence of messages, where each message is calculated from node-local information. Every VMP message update corresponds to a coordinate descent step on the variational free energy, and therefore multiple passes through the schedule converge the free energy to a local minimum.

3. Toolbox-based automated design of message passing algorithms

Message passing provides a convenient paradigm for automating the design of (Bayesian) inference algorithms. A variety of (approximate) inference algorithms can be formulated in terms of message passing, including exact Bayesian inference (sum-product message passing, belief propagation), variational Bayes (VMP), expectation maximization (a special case of VMP), expectation propagation and Gibbs sampling. In principle, these inference algorithms can be generated automatically by finding appropriate message passing schedules and using a library of pre-derived message update equations. This section introduces ForneyLab, a newly developed open source toolbox for automatic generation of inference algorithm based on this paradigm. After a description of the toolbox, we demonstrate its core functionalities through an example application.

3.1. ForneyLab: a toolbox for automating message passing-based inference

To facilitate the automatic generation of message passing solutions to inference problems, we developed ForneyLab (https://github.com/biaslab/ForneyLab.jl), which at the moment of writing this paper is released at version 0.8. ForneyLab is written in the open source scientific programming language Julia, which enjoys a MATLAB-like syntax and native speed similar to compiled *C* code [19]. ForneyLab accepts a specification of the probabilistic model and inference task as inputs, and produces interpretable source code for the desired inference algorithm, thus enabling users to inspect, modify and debug the (message passing-based inference) implementation. This is achieved by scheduling messages on an FFG representation of the model at hand, and using a library of built-in update rules.

Constructing a message passing algorithm with ForneyLab consists of two main steps: specifying the probabilistic model, and defining an inference problem on this model. For the model definition step, ForneyLab provides a convenient domain-specific syntax that resembles notational conventions of alternative probabilistic programming languages. Under the hood, ForneyLab builds the corresponding FFG. Optionally, custom factor nodes and message update rules can be defined outside the framework and re-used in model construction. A node function may internally be represented by a factor graph itself, giving rise to so-called *composite* nodes. This type of support for structural abstraction may be convenient for hierarchical model construction and is also beneficial for algorithmic efficiency. Composite nodes are discussed in more detail in Sec. 4.4.

Once the model has been defined, the user specifies an inference task, which may correspond to (for instance) a signal processing task (by online state estimation) or a parameter estimation task. In general, inference involves finding the marginal posterior distributions of a subset of the model variables. ForneyLab uses the FFG representation of the probabilistic model to automatically derive a suitable message passing schedule, i.e., a sequence of message updates that realizes the requested inference task, and generates source code that implements this message passing algorithm. More specifically, the inference algorithm generation pipeline involves the following stages:

1. **Scheduling**.
   This step corresponds to finding a sequence of message updates that yields all required marginal distributions.
2. **Update rule selection and message type inference**.
   At each factor node, multiple message update rules may be available. ForneyLab chooses the most appropriate rule based on inbound message types and outbound message requirements. ForneyLab contains a library of pre-computed update rules for built-in nodes, but can also use custom, user-defined update rules (see Sec. 4.4 for an example).
3. **Code generation**.
   The sequence of message updates is compiled to source code. Currently, ForneyLab comes with a Julia code generator, but additional code generation engines can be added, for example to generate *C* code or to target computational frameworks like TensorFlow.

Splitting the algorithm generation process into separate stages allows the user to inspect and modify intermediate constructs. For example, it is possible to use a handcrafted schedule instead of an automatically generated one, or to manually change which update rule is applied for a certain message. Since the final result is inference source code, the user is free to probe or manually modify the inference algorithm at any level.

In summary, ForneyLab consists of the following four components: (i) a convenient domain-specific syntax for specifying probabilistic models; (ii) a library of commonly used factor nodes and corresponding message update implementations; (iii) automatic algorithm generators for belief propagation, variational message passing and expectation propagation; and (iv) an “algorithm-to-code” compiler which generates readable and debuggable inference source code. To demonstrate how the algorithm generation process works in practice, we proceed by working out an example.

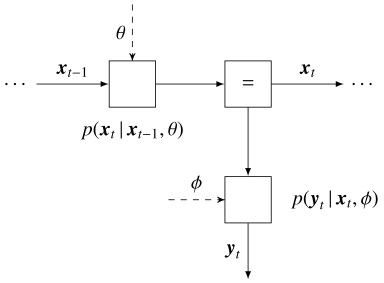

Fig. 4. Factor graph representation of the state-space model of Eq. (5) (priors not drawn). The dots on the left and right sides indicate a repetition of the model section over time. Dashed edges indicate time-independent parameters that are equality constrained over time.

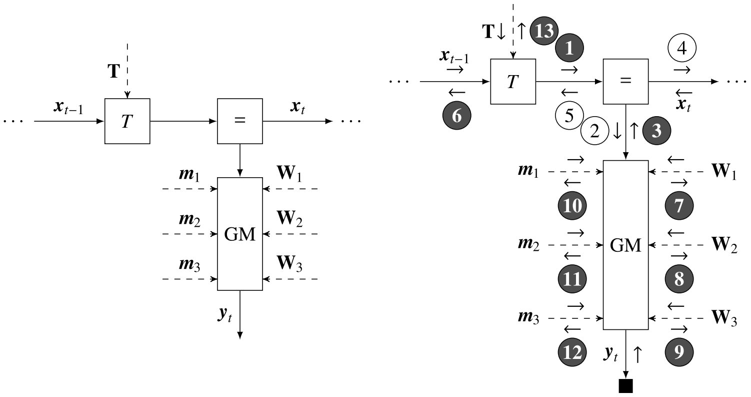

Fig. 5. Factor graph representation and VMP schedule for the HMGM model with $J = 3$ components. Black-labeled messages are computed by the VMP update rule from [16] and white-labeled messages are calculated according to sum-product update rules.

3.2. Example: automated inference in a hidden Markov model with Gaussian mixture emissions

Message passing on factor graphs is known to support a very wide array of statistical signal processing, communication and control engineering algorithms, including Kalman filtering/smoothing, hidden Markov model learning, Viterbi decoding etc. [20]. Most of these algorithms can be interpreted as inference tasks on probabilistic *state-space models* (SSM). In the following, we focus the discussion on automated toolbox-based derivation of inference tasks in probabilistic state-space models. A state-space model (SSM) is defined as

$$p(\boldsymbol{y}, \boldsymbol{x}, \theta, \phi) = \underbrace{p(\boldsymbol{x}_0)\, p(\theta)\, p(\phi)}_{\text{priors}} \prod_{t=1}^{T} \underbrace{p(\boldsymbol{x}_t \,|\, \boldsymbol{x}_{t-1}, \theta)}_{\text{state transition}} \; \underbrace{p(\boldsymbol{y}_t \,|\, \boldsymbol{x}_t, \phi)}_{\text{observation}}, \tag{5}$$

where $\boldsymbol{x} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$ are hidden (unobserved) states, $\boldsymbol{y} = (\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T)$ are observed variables and $\{\theta, \phi\}$ collect the model parameters. In an FFG, time-independent model parameters are constrained to be equal over time by a chain of equality constraint nodes (one for each model section). In order to avoid cluttering the graph with equality chains, we denote these time-independent model parameters by dashed edges in the FFG. The general SSM is graphically represented by the FFG of Fig. 4.

As an example, we consider an SSM that combines a first-order Markov transition model for a discrete state (Eq. (6a)) with a continuously-valued Gaussian mixture observation model (Eq. (6b)). These types of models are successfully used in many applications such as the modeling of fabrication processes, behavioral data and speech signals. Specifically, we consider the model specified by

$$p(\boldsymbol{x}_t \,|\, \boldsymbol{x}_{t-1}, \mathbf{T}) = \text{Categorical}\,(\boldsymbol{x}_t \,|\, \mathbf{T}\boldsymbol{x}_{t-1})\,, \tag{6a}$$

$$p(\boldsymbol{y}_t \,|\, \boldsymbol{x}_t, \boldsymbol{m}, \mathbf{W}) = \prod_{k=1}^{K} \mathcal{N}\left(\boldsymbol{y}_t \,\middle|\, \boldsymbol{m}_k, \mathbf{W}_k^{-1}\right)^{x_{t,k}}, \tag{6b}$$

where $\boldsymbol{x}_t$ is a one-hot coded vector representing the hidden state at time t, and $\mathbf{T}$ is the state transition probability matrix. The vector $\boldsymbol{y}_t \in \mathbb{R}^d$ holds the observations, and $\{\boldsymbol{m}_k, \mathbf{W}_k\}$ are the parameters for the k-th mixture component. The FFG for the generative model with 3 components is drawn in Fig. 5 (left). In contrast to a standard Gaussian mixture model

```
# Priors
@RV T ~ Dirichlet(ones(3,3))
@RV m1 ~ GaussianMeanVariance(zeros(2), huge*diageye(2))
@RV W1 ~ Wishart(huge*diageye(2), 2.0)
...
@RV x_0 ~ Categorical(1/3*ones(3), id=:x_0)

x = Vector{Variable}(n_samples) # Pre-allocate variable vector
y = Vector{Variable}(n_samples)
x_t_min = x_0 # Initialize previous state
for t = 1:n_samples # Build model sections
    @RV x[t] ~ Transition(x_t_min, T) # Transition model
    @RV y[t] ~ GaussianMixture(x[t], m1, W1, m2, W2, m3, W3) # Observation model

    x_t_min = x[t] # Reset state for next section

    placeholder(y[t], :y, index=t, datatype=Float64, dims=(2,)) # Indicate observation of y at time t
end
```

Fig. 6. Julia code for building the HMGM model from Eqs. 6a, 6b with ForneyLab. In Julia, expressions prefixed by "@" indicate macros. Here, `@RV` describes a "Random Variable"-node constructor. The ":" prefix (e.g. `:y`) identifies a symbol that may be used for indexing. The `placeholder` function marks a model variable as observed.

(GMM), this model includes a time-dependent discrete state vector that identifies the mixture component from which the observations are drawn. Furthermore, the model differs from a standard hidden Markov model (HMM) because it includes a more complex emission model for continuously-valued observations (rather than discrete observations). We refer to this model as a Hidden Markov Gaussian Mixture (HMGM) model. In ForneyLab, the HMGM model is specified by the code fragment shown in Fig. 6.

Given a sequence of observations $\boldsymbol{y} = \big(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_T\big)$, we are interested in estimating both the hidden state sequence $\boldsymbol{x}$ and the model parameters $\{\mathbf{T}, \boldsymbol{m}, \mathbf{W}\}$. In other words, we wish to compute

$$p(\boldsymbol{x}, \mathbf{T}, \boldsymbol{m}, \mathbf{W} \,|\, \boldsymbol{y}) = \frac{p(\boldsymbol{y}, \boldsymbol{x}, \mathbf{T}, \boldsymbol{m}, \mathbf{W})}{\sum_{\boldsymbol{x}} \iiint p(\boldsymbol{y}, \boldsymbol{x}, \mathbf{T}, \boldsymbol{m}, \mathbf{W})\, \mathrm{d}\mathbf{T}\, \mathrm{d}\boldsymbol{m}\, \mathrm{d}\mathbf{W}} .$$

The integrals in the denominator are not analytically tractable, making it impossible to perform exact inference, for example through sum-product message passing. However, it is possible to find an approximate solution to the inference problem by resorting to the variational Bayes algorithm, which we will do here. For the variational approximation to the true posterior, we choose the following factorization:

$$q(\boldsymbol{x}, \mathbf{T}, \boldsymbol{m}, \mathbf{W}) = q(\boldsymbol{x})\, q(\mathbf{T}) \prod_{k=1}^{K} q(\boldsymbol{m}_k)\, q(\mathbf{W}_k) . \tag{7}$$

This factorization is known as a *structured variational approximation* since $q(\boldsymbol{x})$ – the approximate posterior distribution of the hidden state sequence – does not fully factorize over all time-indexed state variables, i.e., we do not assume $q(\boldsymbol{x}) = q(\boldsymbol{x}_0)q(\boldsymbol{x}_1)\cdots q(\boldsymbol{x}_T)$. Given the factorization of Eq. (7), the optimal analytic forms of the factors of q are determined by the generative model, so there is no need to specify these forms manually.

Fig. 7 lists the code to specify the recognition distribution and build the corresponding inference algorithm with ForneyLab. The resulting message passing schedule is illustrated in Fig. 5 (right) and involves both VMP (black-labeled) and sum-product (white-labeled) update rules. Where sum-product messages perform exact (Bayesian) updates, the VMP messages introduce approximations by directly using the recognition distributions in their update computations. For details, we refer to the detailed description of VMP on FFGs by Dauwels [16].

The message passing schedule should be repeated until the free energy has converged to a local minimum. The source code that ForneyLab generates to implement the inference algorithm contains one (Julia) function for each factor in the recognition distribution. These functions update the parameters of the corresponding factor in the recognition distribution. Fig. 8 shows a snippet of the function for updating $q(\boldsymbol{x})$. The complete message passing algorithm in this function contains 249 calls to message update functions. The snippet shows how the computation for `message[1]` depends on observation `data[:y][8]` and the current beliefs over the mixture components, as stored in the `marginals` dictionary. Furthermore, the computation for e.g. `message[10]` depends on `message[9]`, showing the sequentiality of

```julia
q = RecognitionFactorization([x_0; x], T, m1, W1, m2, W2, m3, W3; ids=[:X, :T, :M1, :W1, :M2, :W2, :M3, :W3])
algo = variationalAlgorithm(q) # Build the VMP algorithm
algo_F = freeEnergyAlgorithm(q) # Build algorithm to evaluate the variational free energy
```

Fig. 7. Julia code for specifying and building a variational message passing algorithm with ForneyLab. The first line specifies a structured recognition distribution, where all state variables are accommodated in a single recognition factor (indicated by square brackets), with corresponding id `:X`. Subsequent variables each have their own recognition factor and corresponding ids. The final two lines invoke the ForneyLab algorithm generators that return the variational message passing algorithm and free energy algorithm as Julia source code.

```julia
function stepX!(data::Dict, marginals::Dict=Dict(), messages::Vector{Message}=Array{Message}(249))
    messages[1] = ruleVBGaussianMixtureZCat(ProbabilityDistribution(Multivariate, PointMass, m=data[:y][8]),
        nothing, marginals[:m1], marginals[:W1], marginals[:m2], marginals[:W2], marginals[:m3],
        marginals[:W3])
    ...
    messages[10] = ruleSVBTransitionOutVCD(nothing, messages[9], marginals[:T])
    ...
    marginals[:x_0] = messages[9].dist * messages[249].dist
    marginals[:x_1] = messages[10].dist * messages[248].dist
    ...
    return marginals
end
```

Fig. 8. Segment of an automatically generated inference algorithm code as stored in `algo` (Fig. 7). For variational message passing, each recognition factor has its own corresponding `step!` function, where the trailing characters relate to the corresponding recognition factor, e.g. `stepX!`. Each `step!` function consecutively builds an array of messages by executing specific message update rules. In this example, the first message is computed by calling the update `ruleVBGaussianMixtureZCat` on a data entry (`data[:y][8]`) and a pre-initialized dictionary of marginal beliefs. Finally, an updated dictionary of marginal beliefs is returned.

```julia
function freeEnergy(data::Dict, marginals::Dict)
    F = 0.0
    F += averageEnergy(Dirichlet, marginals[:T], ProbabilityDistribution(MatrixVariate, PointMass,
        m=[1.0 1.0 1.0; 1.0 1.0 1.0; 1.0 1.0 1.0]))
    ...
    F += averageEnergy(GaussianMixture, ProbabilityDistribution(Multivariate, PointMass, m=data[:y][1]),
        marginals[:x_1], marginals[:m1], marginals[:W1], marginals[:m2], marginals[:W2], marginals[:m3],
        marginals[:W3])
    ...
    F -= differentialEntropy(marginals[:T])
    F -= differentialEntropy(marginals[:m1])
    ...
    return F
end
```

Fig. 9. Segment of automatically generated code for evaluating the variational free energy as stored in `algo_F` (Fig. 7). The free energy is computed by summing and subtracting (local) energy and entropy terms (see also Eq. (4a)).

the message passing algorithm. In the end, the resulting marginals, e.g. `marginals[:x_0]`, are updated by multiplying colliding messages. These marginals are then used in the updates for the other recognition factors. Inspection of the (also automatically generated) algorithm for evaluating the variational free energy (Fig. 9) reveals that the free energy is also computed by a sequence of node-local computations. As discussed in Sec. 2.2, the value of the minimized free energy functional can be used as a performance metric for the generated algorithm, since minimized free energy is a proxy for "minus log-evidence".

Fig. 10 shows example code for executing the generated inference algorithm. This involves setting the initial recognition distributions, collecting observed variables and repeatedly executing the algorithm until convergence. To test the algorithm, we apply it on a synthetic data set sampled from the generative model with fixed parameters. Fig. 11 (left) shows the data set, which contains 50 two-dimensional observations drawn from a HMGM model with $K = 3$ components. Fig. 11 (middle) visualizes the inferred model parameters (means of the approximate posterior distributions) after convergence of the inference algorithm. The solution correctly identifies all Gaussian mixture components in the observation model, and finds appropriate state transition probabilities for the hidden Markov model that governs the component switches. The

```
# Load algorithms (algo and algo_F are strings containing the source code)
eval(parse(algo))
eval(parse(algo_F))

# Initial recognition distributions
marginals = Dict{Symbol, ProbabilityDistribution}(
    :T  => vague(Dirichlet, (3,3)),
    :m1 => ProbabilityDistribution(Multivariate, GaussianMeanVariance, m=[0.0, 1.0], v=100.0*diageye(2)),
    :W1 => ProbabilityDistribution(Wishart, v=10.0*diageye(2), nu=2.0),
    ...
)

# Initialize data
data = Dict(:y => y_data)
n_its = 20

# Execute algorithm by iteratively calling automatically generated functions
F = Vector{Float64}(n_its)
for i = 1:n_its
    stepX!(data, marginals)
    stepW1!(data, marginals)
    ...
    stepM1!(data, marginals)
    ...
    stepT!(data, marginals)

    F[i] = freeEnergy(data, marginals)
end
```

Fig. 10. Julia code snippet for executing the VMP algorithm generated by ForneyLab. The first two lines of code parse and evaluate the automatically generated algorithm code from Fig. 7. This imports the recognition factor-specific `step!` functions (Fig. 8) and the `freeEnergy` function (Fig. 9) into the current scope. The `step!` functions can then be iterated to update (in-place) a pre-initialized dictionary of marginal beliefs. After each iteration the free energy can be computed and inspected for convergence.

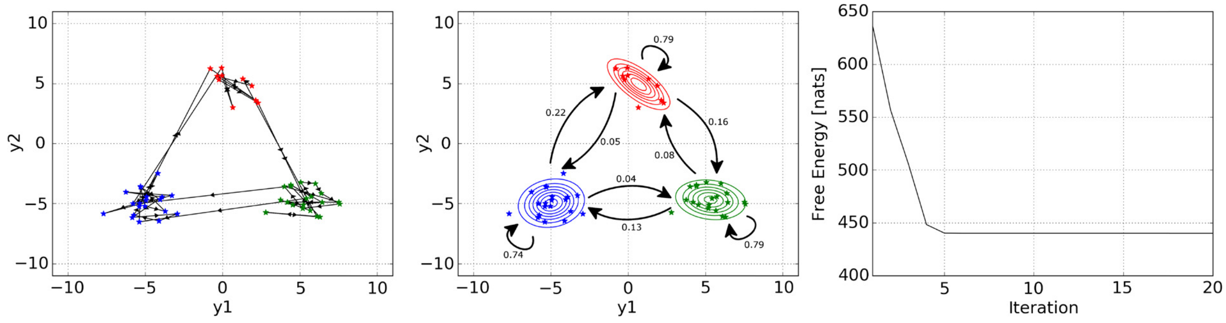

Fig. 11. Left: synthetic two-dimensional data set generated by sampling a sequence of 50 time steps from a HMGM model with $K = 3$ components. Colors correspond to the latent hidden state of the Markov model, arrows indicate the sequence of observations. The parameters of the generative model are chosen to only allow 'clockwise' state transitions. Middle: visualization of the inferred model parameters (Gaussian mixture model and state transition probabilities) after convergence of the variational Bayes algorithm. Right: evolution of the variational free energy during execution of the inference algorithm. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

variational free energy converges to a local minimum after roughly 5 iterations of the VMP algorithm, as shown by Fig. 11 (right).

The automatically generated algorithm for full Bayesian inference in the HMGM model is an illustrative example of how the message passing paradigm can help to produce efficient algorithms without requiring expert knowledge about the underlying methods. While it is possible to manually derive a variational inference algorithm for the model at hand, it involves long and tedious (model-dependent) derivations. Using the divide-and-conquer approach of message passing, we are able to reduce this task to deriving update rules for individual factor nodes, which is much simpler. Moreover, these update rules can be reused to perform inference in other models involving the same factors. Note that while black-box inference methods like automatic differentiation variational inference (ADVI) [21] require no manual derivations at all, they are generally much slower than (message passing) algorithms that leverage analytic solutions. Moreover, additional tricks are required to make ADVI possible in models involving discrete latent variables (see e.g. [22]). In Sec. 5 we compare message passing-based inference to black-box inference methods in terms of performance.

4. Boosting the algorithm design loop by automated inference in factor graphs

In the search process for an effective algorithm, it is important that we can quickly specify updates to model proposals and compare the relative performance of these proposals. Here, we illustrate how the modularity of the FFG framework allows for flexible construction of custom models and algorithms. In Sec. 4.1 we model a time series with a linear Gaussian model. This model is improved in Sec. 4.2, where we exemplify how the model can be adapted by explicitly incorporating nonlinearities. With ForneyLab, this adaptation evaluates to adding one extra line of code to the model specification. In Sec. 4.3, we showcase the flexibility with respect to algorithm specification by constructing a custom algorithm that combines VMP with expectation propagation. We will see that the combined benefits of both algorithms leads to improved performance over pure VMP in our example. Finally, Sec. 4.4 underlines the hierarchical nature of the FFG framework by considering *composite* nodes as modular building blocks for model construction. This construct allows for building complex hierarchical structures, and improved inference by leveraging external tools and custom updates.

4.1. Time series modeling with a linear Gaussian SSM

In this section we provide an example of a linear Gaussian SSM that we will later adapt (Sec. 4.2) to improve the model fit to a simulated data set. Consider a time series data set consisting of hourly temperature measurements over a period of two days generated by the following process:

$$\hat{\boldsymbol{w}}_t \sim \mathcal{N}\left(\mathbf{0}, \hat{\mathbf{W}}^{-1}\right) \tag{8a}$$

$$\hat{\boldsymbol{x}}_t = \mathbf{A}\hat{\boldsymbol{x}}_{t-1} + \hat{\boldsymbol{w}}_t \tag{8b}$$

$$\hat{v}_t \sim \mathcal{N}(0, \hat{u}^{-1}) \tag{8c}$$

$$\hat{y}_t = \log\left(1 + \exp\left(\boldsymbol{b}^{\mathrm{T}}\hat{\boldsymbol{x}}_t\right)\right) + \hat{v}_t\,. \tag{8d}$$

In order to make things interesting, we assumed that our thermometer has a nonlinear response curve, given by the softplus function $g(x) = \log(1 + \exp(x))$, which truncates negative temperatures. In order to introduce a daily periodicity in the temperatures, the hidden state represents a phasor $\phi_t \in \mathbb{C}$, of which the real and imaginary component are respectively encoded by the entries in the hidden states $\hat{\boldsymbol{x}}_t \in \mathbb{R}^2$. Two days ($T = 48$) of data are generated from an initial state $\hat{\boldsymbol{x}}_0 = (5, 0)^{\mathrm{T}}$ by recursive application of Eq. (8a)–(8d). The state transition matrix $\mathbf{A}$ represents a rotation with angular frequency $\pi/12$ and $\hat{\mathbf{W}}$ is a diagonal precision matrix with $\hat{W}_{jj} = 50$, $\hat{W}_{i \neq j} = 0$. The observation is generated from the first (real) state vector component by the selection vector $\boldsymbol{b} = (1, 0)^{\mathrm{T}}$. Finally, $\hat{u} = 50$ represents the precision of the Gaussian observation noise. The resulting observations and hidden states are shown in Fig. 13.

In order to model this time series we postulate a linear Gaussian dynamic model as defined by

$$p(\boldsymbol{x}_t \,|\, \boldsymbol{x}_{t-1}, \mathbf{W}) = \mathcal{N}\left(\boldsymbol{x}_t \,\middle|\, \mathbf{A}\boldsymbol{x}_{t-1}, \mathbf{W}^{-1}\right) \tag{9a}$$

$$p(y_t \,|\, \boldsymbol{x}_t, u) = \mathcal{N}\left(y_t \,\middle|\, \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x}_t, u^{-1}\right), \tag{9b}$$

with “vague” (virtually uninformative) priors on the initial state and precisions. Because this generative model for the data lacks the softplus nonlinearity that was used in generating the data, there exists a deliberate mismatch between the generative process and our proposed generative model. In Sec. 4.2 we will alleviate this mismatch by explicitly modeling the nonlinearity as well.

Assume that we are interested in estimating a posterior belief for the hidden state sequence ($\boldsymbol{x}$), and the transition and observation noise precisions ($\mathbf{W}$ and u) from a given data set ($y = \hat{y}$). In other words, we are interested in evaluating the inference task

$$p(\boldsymbol{x}, \mathbf{W}, u \,|\, y) = \frac{p(y, \boldsymbol{x}, \mathbf{W}, u)}{\int \cdots \int p(y, \boldsymbol{x}, \mathbf{W}, u)\, \mathrm{d}\boldsymbol{x}\, \mathrm{d}\mathbf{W}\, \mathrm{d}u}\,.$$

In order to evaluate this inference task, we perform approximate inference by variational message passing [16] and choose the recognition distribution factorization

$$q(\boldsymbol{x}, \mathbf{W}, u) = q(\boldsymbol{x})\, q(\mathbf{W})\, q(u)\,,$$

which imposes a *structured* factorization of q by assuming a single joint recognition factor for the full state sequence $\boldsymbol{x} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_T)$, and a single joint recognition factor for all entries of the precision matrix $\mathbf{W}$. The resulting message

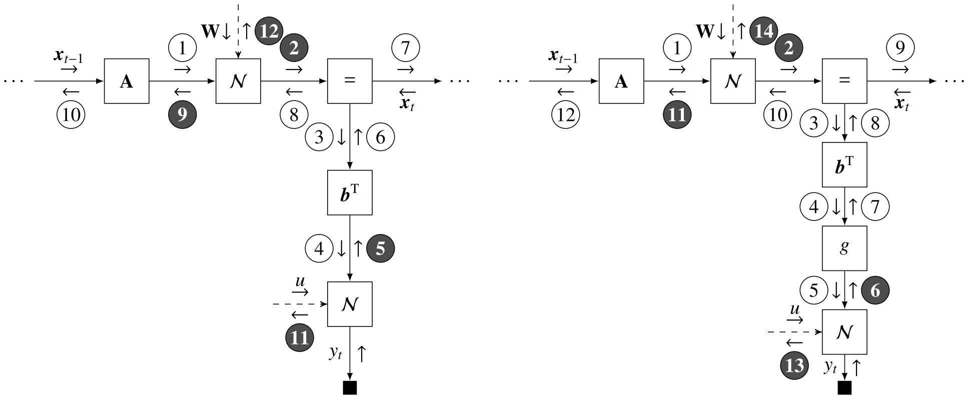

Fig. 12. Message passing schedule (left) for estimation on a linear Gaussian state-space model (see Eqs. (9a), (9b); priors not drawn). Black-labeled messages are computed through the variational message passing update rule from [16]. The right figure shows the message passing schedule for the model with the extended (nonlinear) observation model introduced in Eq. (10).

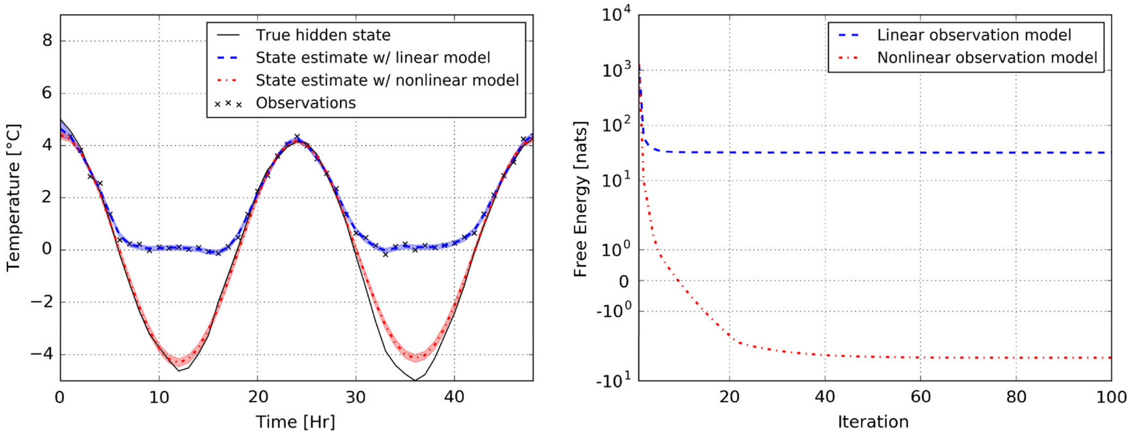

Fig. 13. Inference results for estimation (left) of linear and nonlinear Gaussian state-space models on a toy data set. The right figure shows the convergence of the free energy for both cases.

passing schedule for the current problem is illustrated in Fig. 12 (left). First, messages ① and ❷ estimate a current state from the previous state, and messages ③ and ④ propagate predictions towards the current observation. Next, the current state estimate is corrected by messages ❺, ⑥ and ⑦, which account for evidence contained by the current observation. Propagating forward, these messages execute a forward (filtering) pass over the full state sequence (i.e., going forward from $t = 0$ to $t = T$). This is followed by a backward (smoothing) pass comprising the messages ⑧, ❾ and ⑩ that runs from $t = T$ backwards to $t = 0$. The smoothing pass improves the state estimates through evidence (observations) from future time steps. In a real-time processing scenario, we may need to skip the smoothing pass. Finally, given the updated state estimates, messages ⓫ and ⓬ update the estimates for the precision parameters. This message passing sequence constitutes one iteration of the estimation process. The performance of the estimation process can be improved by repeating the sequence over multiple iterations, where the resulting estimates of an iteration are taken as initial estimates (priors) for the next iteration.

The simulation results for the state estimate after 100 iterations is shown in Fig. 13 (left). It can be seen that the estimated state faithfully follows the observed data. Posterior precisions are $q(u) = \mathrm{Gam}\,(u \,|\, a = 25, b = 0.36)$, and $q(\mathbf{W}) = \mathcal{W}\left(\mathbf{W} \,\middle|\, \mathbf{V} = \begin{pmatrix} 3.87 & 0.51 \\ 0.51 & 0.09 \end{pmatrix}, \nu = 50\right)$ (where $\mathcal{W}$ denotes a Wishart distribution). Fig. 13 (right) shows the free energy as a function of number of iterations.

4.2. Trying an alternative model with a nonlinear likelihood

We now try to improve the algorithm performance of the SSM by postulating an alternative observation model. Specifically, we will explicitly account for the nonlinear corruption of the measurements. Without an automated-inference toolbox, any model adaptation would require a possibly tedious manual re-derivation of the inference update equations. In the factor graph framework, the probabilistic model can be readily extended, and the adjusted VMP algorithm can be automatically de-

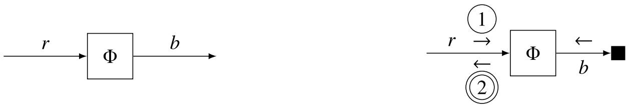

Fig. 14. FFG representation (left) and message passing schedule with observed datum (right) for a binary node. Here, the double circled message represents an expectation propagation message [25].

rived. With ForneyLab, this extension evaluates to a single extra line in the model definition. With the softplus nonlinearity $g(\cdot)$ included, the observation model becomes

$$p(y_t\,|\,\boldsymbol{x}_t, u) = \mathcal{N}\left(y_t \,\middle|\, g(\boldsymbol{b}^{\mathrm{T}}\boldsymbol{x}_t), u^{-1}\right) . \tag{10}$$

The message passing schedule for the nonlinear model is drawn in Fig. 12 (right). Exact computation of message ⑤ is complicated by the nonlinearity introduced by the g node. However, message passing allows us to retain conjugacy by computing messages through local approximations. Therefore, we linearize $g(\cdot)$ around the mean of the inbound message ④, and compute the outbound message from the approximated node function. The same procedure is applied to message ⑦. While this local approximation introduces an error in the resulting message computations, we will see that it works well in practice.

Estimation of the alternative model with ForneyLab yields the state (first component) estimation result of Fig. 13 (left), precision estimates $q(u) = \mathrm{Gam}\,(u\,|\,a = 25, b = 0.50)$ and $q(\mathbf{W}) = \mathcal{W}\left(\mathbf{W} \,\middle|\, \mathbf{V} = \begin{pmatrix} 5.65 & 0.00 \\ 0.00 & 5.62 \end{pmatrix}, \nu = 50\right)$, and free energy estimate in Fig. 13 (right). The final difference in free energy between the linear and nonlinear model evaluates to 156 [dB], which rules overwhelmingly in favor of the nonlinear model.

This section intended to exemplify the ease with which an alternative model proposal can be scored on performance. By simply replacing or adding factor nodes, alternative models can be specified and ForneyLab automatically delivers code for inference algorithms, including code for performance evaluation. This eliminates the need for tedious manual derivations, and opens up the possibility of fast iterative search for the best model fit to the data [1]. Furthermore, we illustrated how message passing allows for local approximations to difficult messages resulting from nonlinearities in the model.

4.3. Message passing on FFGs as a platform for combining inference algorithms

In this section we exemplify how message passing with ForneyLab combines VMP with expectation propagation (EP) for estimating an SSM with a binary observation model. Similar models are often used in the context of perception and decision making, e.g. [23,24]. Here we exemplify how a hybrid VMP-EP algorithm leads to improved estimation results over full VMP on a binary model example.

In this example, we use the same hidden state data generating process as in Sec. 4.1. We generate a hidden state sequence by Eqs. (8a) and (8b) with initial state $\hat{x}_0 = (1, 0)^{\mathrm{T}}$. Then, in contrast to Sec. 4.1, we draw $T = 96$ binary observations $\hat{y}_t \in \{1, -1\}$ (true, false), where the probability of the outcomes is determined by the fluctuating continuous hidden state, as

$$Pr(\hat{y}_t = 1) = \Phi(\boldsymbol{b}^{\mathrm{T}}\hat{\boldsymbol{x}}_t) , \tag{11a}$$

where $\Phi(\cdot)$ is the standard Gaussian cumulative density function.

For the generative model we assume Eq. (9a) for the state transition model, and define the observation model as

$$p(y_t\,|\,\boldsymbol{x}_t) = \Phi\left(y_t \cdot \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x}_t\right) . \tag{12}$$

The full generative model is obtained by substituting Eqs. (12) and (9a) in Eq. (5), and choosing vague priors.

Assume that we are interested in inferring a posterior belief over the hidden state sequence ($\boldsymbol{x}$) and the transition precision ($\mathbf{W}$) from the observed discrete data set ($y = \hat{y}$). As before, we choose a structured recognition distribution factorization, given by

$$q(\boldsymbol{x}, \mathbf{W}) = q(\boldsymbol{x})\, q(\mathbf{W}) . \tag{13}$$

However, in formulating the message passing algorithm, we immediately run into trouble with the factor $f_\Phi(b, r) = \Phi(b \cdot r)$, linking a binary variable $b \in \{1, -1\}$ to a real variable $r \in \mathbb{R}$. We first zoom in on this binary factor node, which is drawn in Fig. 14 (left).

Naively, we might attempt to compute the backward message (indicated by a left overhead arrow) for an observation $\hat{b} = 1$ as

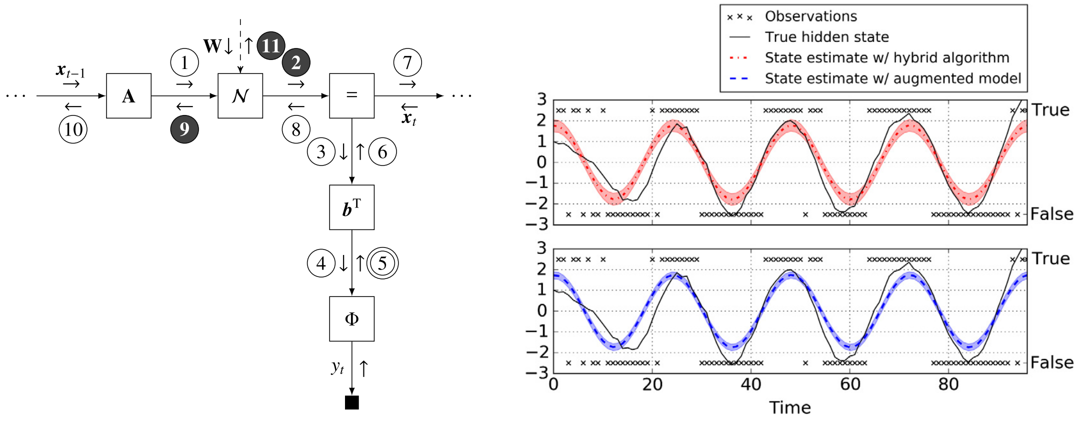

Fig. 15. VMP-EP message passing schedule (left) for estimation on a linear Gaussian SSM with sigmoid observation model, together with a comparison of the hidden state estimates for VMP-EP (top right) and VMP with model augmentation (bottom right). The doubly circled message is computed by the EP update rule [25].

$$\overleftarrow{\mu}(r) = \sum_{b\in\{1,-1\}} f_\Phi(b,r)\,\delta(b-\hat{b}) = \sum_{b\in\{1,-1\}} \Phi(b\cdot r)\,\delta(b-1) = \Phi(r)\,.$$

However, this message breaks conjugacy and leads to increasingly complex messages when propagated further into the model. In an effort to obtain conjugate updates, some proposals mend the variational message updates [26,27], while others augment the generative model [28]. In this section, we propose an alternative approach that takes full advantage of the modularity of the FFG framework. The intrinsic modularity allows for selecting different Bayesian approximation methods at each node-edge interface in the generative model. Thus, the FFG framework allows not only to change local model assumptions, but also to change local inference methods.

Expectation propagation (EP) [10] is an alternative principled approximate Bayesian message passing algorithm that leads to accurate estimates for binary models [29]. For a backward message on the binary node, the EP update is drawn in Fig. 14 (right), where the EP message $\textcircled{2}$ is computed from observation b (which is the incoming message to node Φ from the right), together with a so-called *cavity* message $\textcircled{1}$, which is simply the incoming message to node Φ from the left side of the FFG. Since EP messages introduce circular message dependencies in the schedule, proper EP-based inference is an iterative algorithm, where multiple iterations of the message passing schedule (hopefully) lead to a stable posterior estimate. For more details on the EP message update for message passing on an FFG, see [25]. Using a local EP message at the Φ node naturally leads to a hybrid VMP-EP schedule, as illustrated in Fig. 15 (left).

We compared the performance of the hybrid VMP-EP algorithm with a more conventional model augmentation technique [28] that allows for full VMP estimation. We inferred the posterior results by performing fifty iterations of both algorithms. The resulting state estimates are shown in Fig. 15 (right). The posterior precisions are $\mathcal{W}\left(\mathbf{W}\,\middle|\,\mathbf{V}=\begin{pmatrix}31.5 & 0.0\\ 0.0 & 31.5\end{pmatrix}, \nu=98\right)$ for the hybrid algorithm and $\mathcal{W}\left(\mathbf{W}\,\middle|\,\mathbf{V}=\begin{pmatrix}32.2 & 0.0\\ 0.0 & 32.2\end{pmatrix}, \nu=98\right)$ for the augmented model. In order to assess model performance we compared the terminal free energies, which lean 10 [dB] in favor of the hybrid VMP-EP algorithm.

4.4. Composite nodes for hierarchical model construction and computational efficiency

An important feature of the FFG framework is that a set of neighboring nodes can be grouped together to form a *composite node* that by the rest of the graph is interpreted as a regular single node. Composite nodes hide their internal processing from the rest of the graph, and consequently, inference processes on a graph can proceed as long as each composite node follows the proper message passing communication rules at its interfaces to the rest of the graph.

Composite nodes can be used as modular building blocks in hierarchical model specifications. For instance, a set of nodes can be grouped together as a “layer”-composite node, and a set of connected “layer” nodes can be grouped as a “layered network”-composite node. This “layered network”-composite node can now be inserted at any place in any proper FFG, since composite nodes act as regular nodes at their interfaces. In this view, the FFG framework is “just” a framework for distributed information processing that specifies how different modules in the network communicate with each other. In a sense, FFGs provide the means to do “gray-box” inference in probabilistic models since nodes may hide custom black-box inference procedures.

Since the internal information processing in composite nodes is hidden from the rest of the graph, we can replace parts of the internal message passing computations by other more efficient computations that do not need to be based on message passing. Taking this idea a bit further, it is entirely conceivable to wrap a deep neural network (DNN) from another toolbox into a composite node and use this DNN node as a regular node in our FFG toolbox.

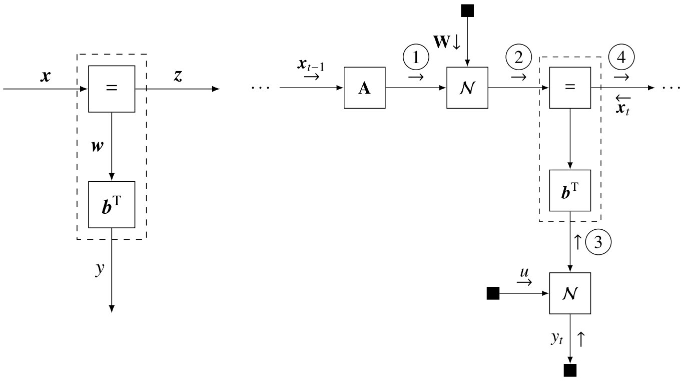

Fig. 16. FFG for the gain-equality composite node (left) and estimation schedule for the linear Gaussian SSM of Sec. 4.1 with a gain-equality composite node included (right).

```
@composite GainEquality (y, x, z) begin
    @RV w = equal(x, z)
    b = [1.0, 0.0]
    @RV y = dot(b, w)
end
```

Fig. 17. Julia code for constructing the gain-equality composite node with ForneyLab. The `@composite` macro header specifies the new node type (`GainEquality`) and an ordered tuple of connected variables `(y, x, z)`. The macro body then defines the (statistical) relations between these (and any internal auxiliary) variables through the standard ForneyLab model definition syntax.

```
@sumProductRule(:node_type      => GainEquality, # Node type the rule pertains to
                :outbound_type => Message{GaussianMeanVariance}, # Resulting message type from update
                :inbound_types => (Message{Gaussian}, Message{Gaussian}, Void), # Argument message types
                :name          => SPGainEqualityIn2GGV) # Unique rule identifier
```

Fig. 18. Julia code for registering a custom gain-equality update rule with ForneyLab. The `@sumProductRule` macro specifies a sum-product update rule by defining an outbound message type for a specific node-message inputs combination. The rule is given a unique name so that its actual computation can be independently implemented by the inference engine.

As an example of the use of composite nodes, we reconsider the SSM of Fig. 12 (left). Here, the update rule for message ⑦ requires the inversion of the covariance matrices of the incoming messages ❷ and ⑥ (see [13]). These inversions might be prohibitively expensive when the dimensionality of the hidden state is large.

The "gain-equality"-composite node, as defined in [13], avoids the need for inverting large covariance matrices by grouping the equality and observation matrix (vector $\boldsymbol{b}$ in our case) into a single node and utilizing the matrix inversion lemma to redefine the message update rule ④ in Fig. 16. As a result, online state estimation in an SSM with large state vectors proceeds more stable and with less computations through the composite node construct. In an FFG, we indicate composite nodes by dashed boxes, see Fig. 16 (left).

ForneyLab provides convenient support to define composite nodes. In order to build this model with ForneyLab, we first define the gain-equality composite node (Fig. 17). This composite node can now be used in the construction of the generative model graph.

If we do nothing else, then the internal message passing in the composite node is the same as without the composite node definition. However, ForneyLab supports creation of custom update rules inside the composite node. For instance, a custom sum-product rule for message ④ in Fig. 16 (right) for the gain-equality node is registered with ForneyLab by the code as shown in Fig. 18.

In summary, composite nodes provide a very powerful mechanism to build hierarchical models and to customize the internal inference processes in these nodes. Customized rules may make use of convenient re-parameterizations, algebraic tricks or sampling methods that could potentially be implemented by external tools and algorithms. The option to leverage external tools for executing message updates also opens up the possibility to *learn* complex updates rules from the data by means of amortization techniques [30,31].

5. Experimental evaluation

In this section we evaluate the usefulness and efficiency of (automatically generated) message passing algorithms for Bayesian inference. To this end, we consider two common scenarios in Bayesian signal processing: Bayesian parameter

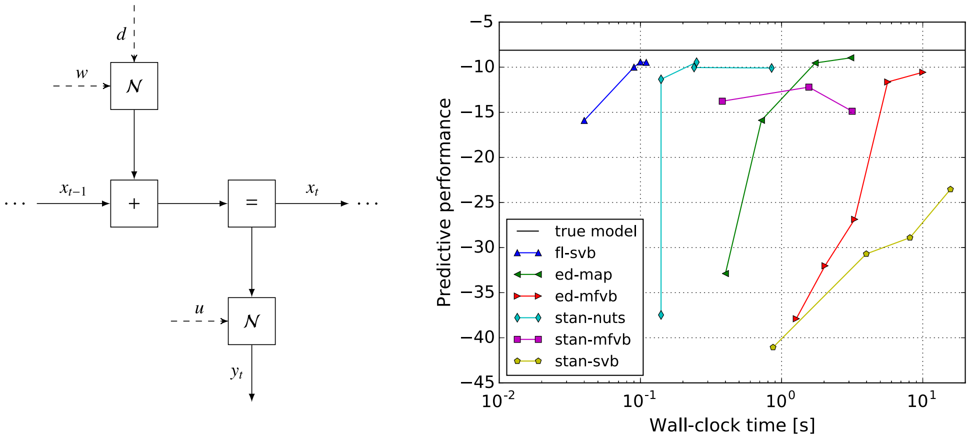

Fig. 19. FFG representation of the considered random walk model (left) and predictive performance vs. running time for multiple inference methods (right) applied to the random walk model. The markers correspond to runs of the respective inference algorithms for varying iteration counts.

estimation in a random walk model (Sec. 5.1) and online learning of the parameters of a linear time-invariant state-space model (Sec. 5.2). We compare the message passing algorithms as implemented with ForneyLab to MCMC and black box variational (ADVI) methods as implemented in probabilistic programming platforms Stan [4,32] and Edward [5].

5.1. Bayesian learning of a random walk model

Learning the parameters of a random walk model with a latent drift component from noisy observations is a common task in signal processing systems. Here we consider the task of full Bayesian estimation of all model parameters, and compare the predictive accuracies and running times of multiple algorithms implemented in ForneyLab, Stan and Edward. Both Monte Carlo and variational algorithms can be viewed as successive approximation methods: the more iterations are performed, the better the list of samples or the variational distribution will approximate the true posterior distribution. Therefore, our goal here is to evaluate the accuracy of multiple algorithm implementations as a function of execution time. Through comparing these performance curves we aim to position ForneyLab in the landscape of automated inference toolboxes in terms of running time versus accuracy.

We consider the following Gaussian random walk model with drift parameter d and Gaussian observation noise:

$$p(x_t \,|\, x_{t-1}, d, w) = \mathcal{N}\left(x_t \,\middle|\, x_{t-1} + d, w^{-1}\right) \tag{14a}$$

$$p(y_t \,|\, x_t, u) = \mathcal{N}\left(y_t \,\middle|\, x_t, u^{-1}\right) . \tag{14b}$$

The left panel of Fig. 19 depicts the FFG representation of this model.

The goal is to perform full Bayesian inference of the hidden state sequence as well as the drift and noise parameters. Initial state x_0 and drift parameter d are endowed with vague Gaussian priors; the priors on the noise precisions are chosen to be vague Gamma distributions. We perform inference using a variety of inference methods and toolboxes:

- No U-Turn Sampling (NUTS, an MCMC method) with Stan (“stan-nuts”);
- Mean-field and full-rank ADVI with Stan (“stan-mfvb” and “stan-svb”);
- MAP inference and mean-field ADVI with Edward (“ed-map” and “ed-mfvb”);
- Structured VMP with ForneyLab (“fl-svb”).

These methods propose different factorizations of the recognition distribution. As a general rule, less factorization assumptions in the recognition distributions is expected to lead to better approximations of the true posterior. On the other hand, a lower degree of factorization also makes it harder to quickly converge to a local minimum in the variational free energy. Full-rank ADVI as performed by Stan assumes no factorization of the recognition distribution. ForneyLab and Edward use the following mean-field and structured factorizations:

$$\text{Mean-field:} \quad q(x, d, w, u) = q(d)\, q(w)\, q(u) \prod_{t=0}^{T} q(x_t) \tag{15a}$$

$$\text{Structured:} \quad q(x, d, w, u) = q(d)\, q(w)\, q(u)\, q(x) . \tag{15b}$$

Inference is performed on a toy data set consisting of 50 samples drawn from a random walk with drift $\hat{d} = -0.1$, transition precision $\hat{w} = 100$ and observation precision $\hat{u} = 10$. For performance evaluation, we draw $N = 1000$ trajectories

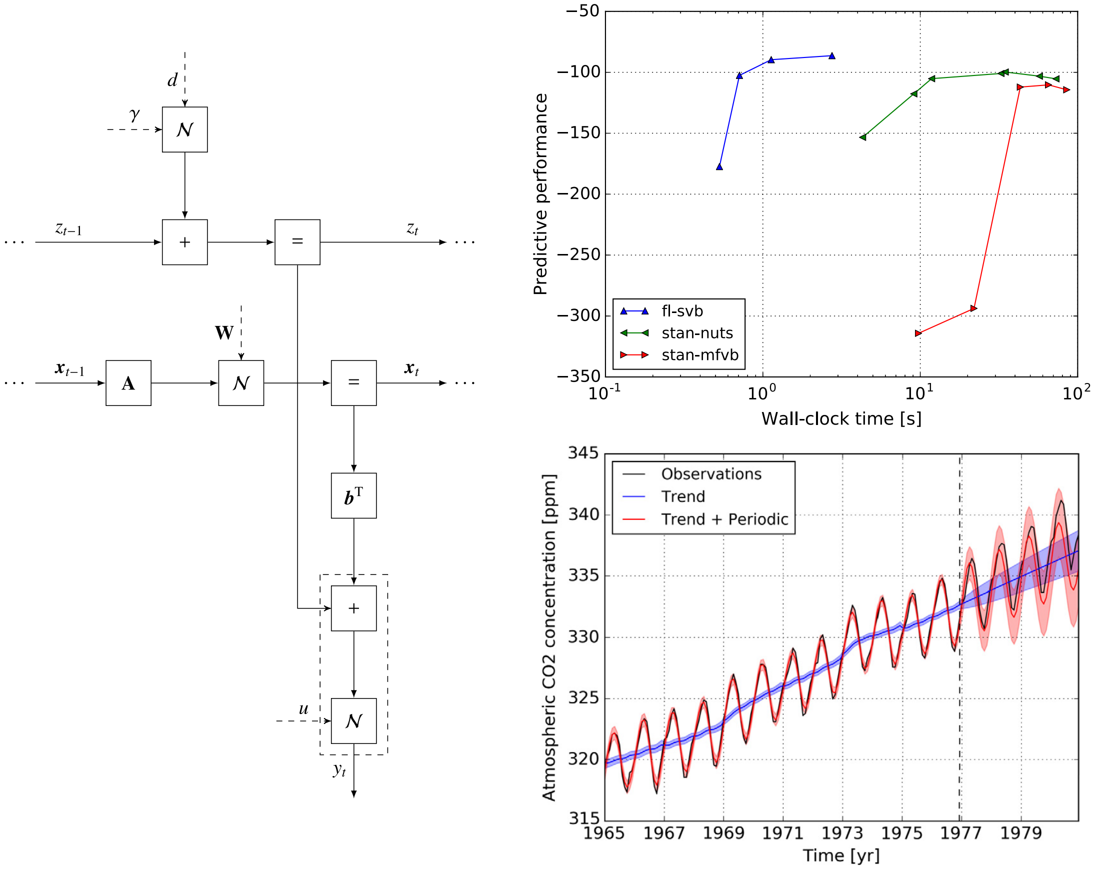

Fig. 20. Left: FFG representation of the model for CO_2 concentration levels. Top right: predictive performance as a function of running time for different inference implementations. Bottom right: CO_2 concentration data set and visualization of the inference result obtained by fl-svb. The shaded area corresponds to two standard deviations; the area to the right of the dashed line corresponds to the predictive distribution of the model under the inferred posterior.

$y_{\text{pred}}^{(n)}$ of length 20 from the true generative process. Then, for each estimation method, we draw $S = 1000$ samples from the (approximate) posterior distribution over the parameters. As a performance measure for the inference procedure, we evaluate the average marginal log-likelihood of each sample-trajectory combination, as defined by

$$Q = \frac{1}{S}\frac{1}{N}\sum_{s=1}^{S}\sum_{n=1}^{N}\log p_s\left(y_{\text{pred}}^{(n)}\right),$$

where p_s is the predictive distribution under sampled parameter setting s. On average, this metric will favor the approximate posterior with the best predictive performance. Fig. 19 (right) depicts predictive performance versus running time for the different inference methods and varying iteration counts. All experiments were executed on the same Linux notebook with 16 GB of memory and no GPU acceleration. Roughly speaking, we see that ForneyLab simulations lead to similar or better predictive performance in less (wall-clock) time.

5.2. Learning a state-space model through streaming variational Bayes

To strain the toolboxes a bit further, in this section we combine the linear Gaussian model of Sec. 4.1 with the random walk model of Sec. 5.1 into a combined SSM, and perform Bayesian inference on a real-world time series. The considered data set is comprised of monthly atmospheric CO_2 concentration measurements [33], which show a (seasonal) periodic component and a slow upward trend, as shown in Fig. 20 (bottom right). The model we consider consists of two components: a Gaussian random walk with drift (to capture the trend) and an SSM for periodic signals (to capture the periodicity). The SSM component models periodicity by applying a rotation matrix **A** in the state transition model, where the latent state $\boldsymbol{x}$ represents a phasor. Finally, the observations are modeled by adding the two components under Gaussian observation noise:

$$\text{Trend model:}\quad p(z_t\,|\,z_{t-1},d,\gamma) = \mathcal{N}\left(z_t\,\middle|\,z_{t-1}+d,\gamma^{-1}\right) \tag{16a}$$

$$\text{Periodic model:} \quad p(\boldsymbol{x}_t \,|\, \boldsymbol{x}_{t-1}, \mathbf{W}) = \mathcal{N}\left(\boldsymbol{x}_t \,\middle|\, \mathbf{A}\boldsymbol{x}_{t-1}\mathbf{W}^{-1}\right) \tag{16b}$$

$$\text{Observation model:} \quad p(y_t \,|\, \boldsymbol{x}_t, z_t, u) = \mathcal{N}\left(y_t \,\middle|\, \boldsymbol{b}^{\mathrm{T}}\boldsymbol{x}_t + z_t, u^{-1}\right) . \tag{16c}$$

Performing full Bayesian inference for the model parameters as well as the hidden state sequences is challenging because of the number of latent variables in the model. A common strategy to keep the computational load limited in such cases is to sequentially perform inference based on mini-batches, using the (approximate) posteriors of the previous step as priors in the next. This approach is known as "streaming variational inference" [34], since it yields an inference algorithm whose computational load scales linearly in terms of data size, thus making it possible to process data in a streaming fashion. In this experiment we apply streaming variational inference with mini-batch size 24. The initial priors are set to be vague. We use the first six mini-batches for learning and the remaining two mini-batches for evaluating the predictive performance of the fitted model. The predictive performance is defined as the average log-likelihood of the test set under 100 samples of the (approximate) posterior.

We compare the same combinations of toolboxes and inference methods as in Sec. 5.1, with the exception of MAP inference since it cannot be applied in a recursive fashion. Unfortunately, we were not able to obtain a converging algorithm in all cases, even after fixing one or more of the model variables and applying informed initializations. In particular, we were unable to construct converging black box variational inference algorithms in Edward for the given model. The ADVI implementation in Stan does converge, but only in case of mean-field factorization. Fig. 20 (top right) contains the results for the methods that converged.

Again, these performance results support the notion that message passing-based inference in factor graphs (as implemented by ForneyLab) is a competitive probabilistic modeling strategy for streaming data applications. In our opinion, when dealing with time-constrained inference and learning problems in dynamical models, message passing-based inference should be a strong candidate inference strategy.

6. Related work

Interest of the machine learning community in probabilistic programming toolboxes has exploded over the last few years. This rise in popularity is catalyzed by the development of TensorFlow [35] as a basis for recent toolboxes such as Edward [5] and ZhuSuan [36]. These toolboxes exploit the analytic expressions of the free energy functional as their optimization objectives. In practice, assumptions about conjugacy or the form of the recognition distribution may limit the scope of workable models. In contrast, sampling-based toolboxes such as PyMC3 [37] (based on Theano [38]) and Stan [4] are in general more flexible in their available modeling choices, but pay a price in terms of estimation time.

Modularity in terms of model specification and computation is another important topic in probabilistic programming, because it allows for efficient re-use of pre-specified model and inference primitives as well as for custom extensions upon existing building blocks. Already from the early beginnings, with the development of the BUGS project, the importance of modularity and extensibility of probabilistic programming toolboxes was recognized [2]. More recent examples of principled approaches to modular model specification are the Bayes Blocks toolbox [39], and the model fragments approach to VMP-based semi-parametric regression [40].

Recently, more effort has been directed towards the positioning of probabilistic programming toolboxes in the scientific process. With the development of Edward, the emphasis of the probabilistic programming toolbox shifts from being just a tool for inference, to it being a tool for model criticism as well. This enables probabilistic programming to aid with the full process of iterative model design [1].

ForneyLab employs the Forney-style factor graph [12] framework for model representation, and implements VMP as described by [16]. VMP was originally described on Bayesian networks by [9] and found an implementation in the VIBES framework [41]. Later implementations of VMP were based on (bipartite) factor graphs [42], which found implementations in Infer.NET [43], Dimple [44] and Bayes Net [45].

Bayes Net and Dimple are both based on the MATLAB language, which is practical, but suffers from performance issues for larger models (and MATLAB itself is not free). In contrast, ForneyLab is based on the modern, open source, high-productivity and high-performance language Julia [19]. Moreover, ForneyLab differs from existing probabilistic programming frameworks because it uses Julia's meta-programming capabilities to produce inference algorithms as executable (Julia) source code. This source code may be readily customized, offering the user precise low-level control over inference execution.

7. Discussion and conclusions

In this paper we approached Bayesian inference from a message passing perspective, based on a Forney-style factor graph (FFG) description of probabilistic generative models. We painted a broad spectrum of possibilities that arise naturally when adhering to FFGs as a platform for Bayesian inference:

- in contrast to ADVI methods, message passing naturally allows for estimation of hybrid models, combining discrete and continuous latent variables (Sec. 3.2);

- the modular make-up of the FFG allows for automated derivation of message passing algorithms (Sec. 4.1);
- the automatically derived free energy functional allows for evaluation of the free energy as a model performance measure (Sec. 4.1);
- the modular model representation, together with a principled and automatically derived performance measure, allow for fast iterative search for the best model to fit the data (Sec. 4.2);
- the message passing formalism allows for combining conceptually distinct inference algorithms under a unified paradigm (Sec. 4.3);
- composite nodes allow for implementing custom rules for improved efficiency and flexibility, and allow for hierarchical design in terms of model structure and algorithms (Sec. 4.4).

We showed proofs of principle for each of these benefits by implementing examples with ForneyLab, a novel probabilistic programming toolbox for message passing on FFGs.

Tran and Blei summarized four future challenges for message-passing based probabilistic programming languages, in their comment [46] on Wand [40]. With ForneyLab, we have addressed (at least in part) these four challenges. The first challenge concerns the specification of local structure by a probabilistic programming language. With ForneyLab, local structure is naturally represented as connections between nodes in an FFG. The user specifies the model through a domain-specific syntax that mimics probabilistic notation, while under the hood the FFG is constructed.

Tran and Blei's second challenge concerned building an extensible inference engine. As mentioned in Sec. 4.4, in ForneyLab, inference schedules can be naturally extended by including custom (composite) nodes and message updates. However, the extensibility of the inference engine does not end there. The user is free to write her own custom inference engine that takes as input the schedules produced by ForneyLab. This engine might then allow for efficiently combining message updates, or even compile the schedule for implementation on custom hardware.

The third challenge proposes to push modularity even further by introducing hierarchicality in algorithms. We touched upon this idea in Sec. 4.4, where composite nodes implement custom update rules that may employ any external toolbox or algorithm it can interface with. This allows for building hierarchies of local algorithms. Combinations of inference algorithms were further explored in Sec. 4.3, where the message passing paradigm allowed for proposing a combined VMP-EP algorithm.

Tran and Blei's fourth challenge related to adapting inference to the computational budget of the user. With message passing, updates can be budgeted based on the information contents of incoming messages. For example, when the entropy of incoming messages is high, the information content of the outbound message might not justify the computational expense. With ForneyLab, budgeting mechanisms may be readily incorporated into the message update rules.

We positioned ForneyLab in the probabilistic toolbox landscape by comparing predictive performance and running time of message passing algorithms as implemented with ForneyLab, with similar inference procedures as implemented with Edward and Stan. For the two SSM problems considered in this paper, ForneyLab exhibited similar or better predictive performance in much less (wall-clock) time. Therefore, we surmise that ForneyLab is a candidate probabilistic modeling framework for automated inference in dynamical systems. These systems are particularly prevalent in the signal processing and control theory communities.

7.1. Limitations

The choice of message passing as a platform for probabilistic inference implies some inherent limitations. Message passing relies on the ability of the generative model to factorize into local node functions, which is not always possible. For example, a Gaussian Process model requires a full joint distribution over the modeled (state) variables, disallowing local factorization assumptions on which message passing is based. A second limitation is due to ForneyLab's requirement for tractable messages. With ForneyLab, messages are represented by a message type together with a finite set of parameters (sufficient statistics). While this representation enables efficient computations, it also limits the flexibility of the posterior distribution. An extension of the message representation might allow for implementation of non-parametric message types such as particle lists. Finally, ForneyLab does not use parallelization and vectorization out of the box. However, with message passing there is ample opportunity to apply these techniques. More efficient schedulers or engines that account for parallel structure might significantly increase performance.

7.2. Why Julia?

Julia is a high-level programming language for technical computing that combines *productivity* with *performance*. The MATLAB-like syntax enables fast and productive prototyping, while the just-in-time (JIT) compiler ensures high performance with execution times close to compiled *C* code [19]. These properties make Julia an excellent language choice for probabilistic programming, where abstract model and algorithm definitions must be compiled to fast executable inference algorithms.

As already shown in the examples above, ForneyLab takes advantage of Julia's qualities. Firstly, productive sessions with ForneyLab are facilitated through Julia's excellent meta-programming functionalities. The domain-specific syntax for defining the generative model employs macros that convert the abstract model declaration expressions to specific calls to the node constructors that construct the FFG under the hood. Meta-programming is also extensively used on the other end of the modeling pipeline, where high-level schedules are converted to executable Julia code. In essence, ForneyLab is a Julia program that outputs Julia programs. This is advantageous since the complete open source Julia suite (base language and over 1800 registered packages, see https://pkg.julialang.org) remains seamlessly accessible while working with ForneyLab.

Furthermore, with Julia's multiple dispatch functionality, message computation rules are defined as factor node-specific functions. For varying input message types, a single update rule may yield different outbound message types. Multiple dispatch allows to group separate methods for varying input signatures under a single function name. This greatly improves clarity and effectively implements an innate message lookup-table.

Finally, the open source Julia license makes it a highly accessible language for science and engineering communities across the academic and industrial worlds.

Acknowledgements

The authors gratefully acknowledge the contributions made to ForneyLab by Ivan Bocharov and Anouk van Diepen at the Signal Processing Systems (SPS) group at the Eindhoven University of Technology. Furthermore, we thank Ismail Senoz, Tjalling Tjalkens and Wouter van Roosmalen at the SPS group, and Joris Kraak at GN Hearing for stimulating discussions. Finally, we thank the two anonymous reviewers for their valuable comments. This research was made possible by financial support from GN Hearing.

References

[1] D.M. Blei, Build compute, critique, repeat: data analysis with latent variable models, Annu. Rev. Stat. Appl. 1 (1) (2014) 203–232, https://doi.org/10.1146/annurev-statistics-022513-115657, http://www.annualreviews.org/doi/abs/10.1146/annurev-statistics-022513-115657.

[2] D.J. Lunn, A. Thomas, N. Best, D. Spiegelhalter, WinBUGS – a Bayesian modelling framework: concepts, structure, and extensibility, Stat. Comput. 10 (4) (2000) 325–337, https://doi.org/10.1023/A:1008929526011, https://link.springer.com/article/10.1023/A:1008929526011.

[3] R. Ranganath, S. Gerrish, D. Blei, Black box variational inference, in: PMLR, 2014, pp. 814–822, http://proceedings.mlr.press/v33/ranganath14.html.

[4] B. Carpenter, A. Gelman, M.D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, A. Riddell, Stan: a probabilistic programming language, J. Stat. Softw. 76 (1) (2017), https://doi.org/10.18637/jss.v076.i01, http://www.jstatsoft.org/v76/i01.

[5] D. Tran, A. Kucukelbir, A.B. Dieng, M. Rudolph, D. Liang, D.M. Blei, Edward: a library for probabilistic modeling, inference, and criticism, arXiv preprint arXiv:1610.09787, https://arxiv.org/abs/1610.09787.

[6] T. Lienart, Y.W. Teh, A. Doucet, Expectation particle belief propagation, in: Advances in Neural Information Processing Systems, 2015, pp. 3609–3617, http://papers.nips.cc/paper/5674-expectation-particle-belief-propagation.

[7] W. Jitkrittum, A. Gretton, N. Heess, S.M.A. Eslami, B. Lakshminarayanan, D. Sejdinovic, Z. Szabó, Kernel-based just-in-time learning for passing expectation propagation messages, in: Proceedings of the Thirty-First Conference, Amsterdam, Netherlands, 2015, http://www.auai.org/uai2015/proceedings/papers/235.pdf, 2015.

[8] J. Pearl, Reverend Bayes on inference engines: a distributed hierarchical approach, in: Proceedings of the Second AAAI Conference on Artificial Intelligence, AAAI'82, AAAI Press, Pittsburgh, Pennsylvania, 1982, pp. 133–136, http://www.aaai.org/Papers/AAAI/1982/AAAI82-032.pdf.

[9] J. Winn, C.M. Bishop, Variational message passing, J. Mach. Learn. Res. 6 (Apr. 2005) 661–694, http://www.jmlr.org/papers/volume6/winn05a/winn05a.pdf.

[10] T.P. Minka, Expectation propagation for approximate Bayesian inference, in: Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence, UAI'01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 362–369, http://dl.acm.org/citation.cfm?id=2074022.2074067.

[11] J. Dauwels, S. Korl, H.-a. Loeliger, Particle methods as message passing, in: 2006 IEEE International Symposium on Information Theory, IEEE, 2006, pp. 2052–2056, http://ieeexplore.ieee.org/document/4036329.

[12] G. Forney, Codes on graphs: normal realizations, IEEE Trans. Inf. Theory 47 (2) (2001) 520–548, https://doi.org/10.1109/18.910573, https://ieeexplore.ieee.org/abstract/document/910573.

[13] H.-A. Loeliger, An introduction to factor graphs, IEEE Signal Process. Mag. 21 (1) (2004) 28–41, https://ieeexplore.ieee.org/document/1267047.

[14] S. Korl, A Factor Graph Approach to Signal Modelling, System Identification and Filtering, Ph.D. thesis, Swiss Federal Institute of Technology, Zurich, 2005.

[15] C. Reller, State-Space Methods in Statistical Signal Processing: New Ideas and Applications, Ph.D. thesis, ETH, Zurich, 2012.

[16] J. Dauwels, On variational message passing on factor graphs, in: IEEE International Symposium on Information Theory, 2007, pp. 2546–2550, http://ieeexplore.ieee.org/abstract/document/4557602.

[17] H. Attias, A variational Bayesian framework for graphical models, in: NIPS, vol. 12, 1999, http://papers.nips.cc/paper/1726-a-variational-baysian-framework-for-graphical-models.pdf.

[18] D.M. Blei, A. Kucukelbir, J.D. McAuliffe, Variational inference: a review for statisticians, J. Am. Stat. Assoc. 112 (518) (2017) 859–877, https://doi.org/10.1080/01621459.2017.1285773, https://www.tandfonline.com/doi/full/10.1080/01621459.2017.1285773.

[19] J. Bezanson, A. Edelman, S. Karpinski, V. Shah, Julia: a fresh approach to numerical computing, SIAM Rev. 59 (1) (2017) 65–98, https://doi.org/10.1137/141000671, https://epubs.siam.org/doi/abs/10.1137/141000671.

[20] H.-A. Loeliger, J. Dauwels, J. Hu, S. Korl, L. Ping, F.R. Kschischang, The factor graph approach to model-based signal processing, Proc. IEEE 95 (6) (2007) 1295–1322, https://doi.org/10.1109/JPROC.2007.896497.

[21] A. Kucukelbir, D. Tran, R. Ranganath, A. Gelman, D.M. Blei, Automatic differentiation variational inference, J. Mach. Learn. Res. 18 (1) (2017) 430–474, http://www.jmlr.org/papers/volume18/16-107/16-107.pdf.

[22] G. Tucker, A. Mnih, C.J. Maddison, J. Lawson, J. Sohl-Dickstein, REBAR: low-variance, unbiased gradient estimates for discrete latent variable models, in: Advances in Neural Information Processing Systems, 2017, pp. 2624–2633, http://papers.nips.cc/paper/6856-rebar-low-variance-unbiased-gradient-estimates-for-discrete-latent-variable-models.pdf.

[23] C.D. Mathys, E.I. Lomakina, J. Daunizeau, S. Iglesias, K.H. Brodersen, K.J. Friston, K.E. Stephan, Uncertainty in perception and the hierarchical Gaussian filter, Front. Human Neurosci. 8 (2014) 825, https://doi.org/10.3389/fnhum.2014.00825, http://journal.frontiersin.org/Journal/10.3389/fnhum.2014.00825/full.

[24] S. Bitzer, H. Park, F. Blankenburg, S.J. Kiebel, Perceptual decision making: drift-diffusion model is equivalent to a Bayesian model, Front. Human Neurosci. 8 (2014) 102, https://doi.org/10.3389/fnhum.2014.00102, http://journal.frontiersin.org/Journal/10.3389/fnhum.2014.00102/abstract.

[25] M. Cox, B. de Vries, Robust expectation propagation in factor graphs involving both continuous and binary variables, in: 26th European Signal Processing Conference, 2018, https://biaslab.github.io/pdf/eusipco2018/cox_robust_expectation_propagation.pdf.

[26] D.A. Knowles, T. Minka, Non-conjugate variational message passing for multinomial and binary regression, in: J. Shawe-Taylor, R.S. Zemel, P.L. Bartlett, F. Pereira, K.Q. Weinberger (Eds.), Adv. Neural Inf. Process. Syst., vol. 24, Curran Associates, Inc., 2011, pp. 1701–1709, http://papers.nips.cc/paper/4407-non-conjugate-variational-message-passing-for-multinomial-and-binary-regression.pdf, 2011.

[27] T.H. Nolan, M.P. Wand, Accurate logistic variational message passing: algebraic and numerical details, Stat 6 (1) (2017) 102–112, https://doi.org/10.1002/sta4.139, http://onlinelibrary.wiley.com/doi/10.1002/sta4.139/abstract.

[28] J.H. Albert, S. Chib, Bayesian analysis of binary and polychotomous response data, J. Am. Stat. Assoc. 88 (422) (1993) 669–679, https://doi.org/10.1080/01621459.1993.10476321, https://www.tandfonline.com/doi/abs/10.1080/01621459.1993.10476321.

[29] M. Kuss, C.E. Rasmussen, Assessing approximate inference for binary Gaussian process classification, J. Mach. Learn. Res. 6 (2005) 1679–1704, http://www.jmlr.org/papers/volume6/kuss05a/kuss05a.pdf.

[30] A. Stuhlmüller, J. Taylor, N. Goodman, Learning stochastic inverses, in: Advances in Neural Information Processing Systems, 2013, pp. 3048–3056, http://papers.nips.cc/paper/4966-learning-stochastic-inverses.pdf.

[31] S. Gershman, N. Goodman, Amortized inference in probabilistic reasoning, Proc. Cogn. Sci. Soc. 36 (36) (2014), http://escholarship.org/uc/item/34j1h7k5.

[32] A. Kucukelbir, R. Ranganath, A. Gelman, D. Blei, Automatic variational inference in Stan, in: C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama, R. Garnett (Eds.), Adv. Neural Inf. Process. Syst., vol. 28, Curran Associates, Inc., 2015, pp. 568–576, http://papers.nips.cc/paper/5758-automatic-variational-inference-in-stan.pdf.

[33] K.W. Hipel, A.I. McLeod, Time Series Modelling of Water Resources and Environmental Systems, Elsevier, 1994.

[34] T. Broderick, N. Boyd, A. Wibisono, A.C. Wilson, M.I. Jordan, Streaming variational Bayes, in: Advances in Neural Information Processing Systems, 2013, pp. 1727–1735, http://papers.nips.cc/paper/4980-streaming-variational-bayes.

[35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, TensorFlow: a system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, 2016, pp. 265–283, https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[36] J. Shi, J. Chen, J. Zhu, S. Sun, Y. Luo, Y. Gu, Y. Zhou, ZhuSuan: a library for Bayesian deep learning, arXiv:1709.05870.

[37] J. Salvatier, T.V. Wiecki, C. Fonnesbeck, Probabilistic programming in Python using PyMC3, PeerJ. Comput. Sci. 2 (2016) e55, https://doi.org/10.7717/peerj-cs.55, https://peerj.com/articles/cs-55.

[38] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, Y. Bengio, Theano: a CPU and GPU math compiler in Python, in: Proc. 9th Python in Science Conf., vol. 1, 2010, http://conference.scipy.org/proceedings/scipy2010/pdfs/bergstra.pdf.

[39] M. Harva, T. Raiko, A. Honkela, H. Valpola, J. Karhunen, Bayes blocks: an implementation of the variational Bayesian building blocks framework, arXiv preprint arXiv:1207.1380.

[40] M.P. Wand, Fast approximate inference for arbitrarily large semiparametric regression models via message passing, J. Am. Stat. Assoc. 112 (517) (2017) 137–168, https://doi.org/10.1080/01621459.2016.1197833, https://www.tandfonline.com/doi/full/10.1080/01621459.2016.1197833.

[41] C.M. Bishop, D. Spiegelhalter, J. Winn, VIBES: a variational inference engine for Bayesian networks, in: Advances in Neural Information Processing Systems, 2003, pp. 793–800, http://papers.nips.cc/paper/2172-vibes-a-variational-inference-engine-for-bayesian-networks.pdf.

[42] J.S. Yedidia, W. Freeman, Y. Weiss, Constructing free-energy approximations and generalized belief propagation algorithms, IEEE Trans. Inf. Theory 51 (7) (2005) 2282–2312, https://doi.org/10.1109/TIT.2005.850085, http://ieeexplore.ieee.org/abstract/document/1459044.

[43] T. Minka, J. Winn, J. Guiver, Y. Zaykov, D. Fabian, J. Bronskill, Infer.NET 2.7, Microsoft, Research Cambridge, 2018, http://research.microsoft.com/infernet.

[44] S. Hershey, J. Bernstein, B. Bradley, A. Schweitzer, N. Stein, T. Weber, B. Vigoda, Accelerating inference: towards a full language, compiler and hardware stack, arXiv:1212.2991 [cs, stat], http://arxiv.org/abs/1212.2991.

[45] K. Murphy, The Bayes Net Toolbox for MATLAB, Comput. Sci. Stat. 33 (2) (2001) 1024–1034, http://interfacesymposia.org/I01/I2001Proceedings/KMurphy/KMurphy.pdf.

[46] D. Tran, D.M. Blei, Comment, J. Am. Stat. Assoc. 112 (517) (2017) 156–158, https://doi.org/10.1080/01621459.2016.1270044, https://doi.org/10.1080/01621459.2016.1270044.