Hindawi
Scientific Programming
Volume 2023, Article ID 6601690, 26 pages
https://doi.org/10.1155/2023/6601690

Research Article

Reactive Message Passing for Scalable Bayesian Inference

Dmitry Bagaev and Bert de Vries

Eindhoven University of Technology, Eindhoven, Netherlands

Correspondence should be addressed to Bert de Vries; bert.de.vries@tue.nl

Received 17 February 2023; Revised 30 March 2023; Accepted 20 April 2023; Published 27 May 2023

Academic Editor: Antonio J. Peña

We introduce reactive message passing (RMP) as a framework for executing schedule-free, scalable, and, potentially, more robust message passing-based inference in a factor graph representation of a probabilistic model. RMP is based on the reactive programming style, which only describes how nodes in a factor graph react to changes in connected nodes. We recognize reactive programming as the suitable programming abstraction for message passing-based methods that improve robustness, scalability, and execution time of the inference procedure and are useful for all future implementations of message passing methods. We also present our own implementation ReactiveMP.jl, which is a Julia package for realizing RMP through minimization of a constrained Bethe free energy. By user-defined specification of local form and factorization constraints on the variational posterior distribution, ReactiveMP.jl executes hybrid message passing algorithms including belief propagation, variational message passing, expectation propagation, and expectation maximization update rules. Experimental results demonstrate the great performance of our RMP implementation compared to other Julia packages for Bayesian inference across a range of probabilistic models. In particular, we show that the RMP framework is capable of performing Bayesian inference for large-scale probabilistic state-space models with hundreds of thousands of random variables on a standard laptop computer.

1. Introduction

In this paper, we develop a *reactive* approach to Bayesian inference on factor graphs. We provide the methods, implementation aspects, and simulation results of message passing-based inference realized by a reactive programming paradigm. Bayesian inference methods facilitate the realization of a very wide range of useful applications, but in our case, we are motivated by our interest in the execution of real-time Bayesian inference in state-space models with data streams that potentially may deliver an infinite number of observations over an indefinite period of time.

The main idea of this paper is to combine message passing-based Bayesian inference on factor graphs with a reactive programming approach to build a foundation for an efficient, scalable, adaptable, and robust Bayesian inference implementation. Efficiency implies less execution time and less memory consumption than alternative approaches; scalability refers to running inference in large probabilistic models with possibly hundreds of thousands of random variables; adaptability implies real-time in-place probabilistic model adjustment, and robustness relates to protection against failing sensors and missing data. We believe that the proposed approach, which we call *Reactive Message Passing* (RMP), will lubricate the transfer of research ideas about Bayesian inference-based agents to real-world applications. To that end, we have developed and also present ReactiveMP.jl, which is an open-source Julia package that implements RMP on a factor graph representation of a probabilistic model. Our goal is that ReactiveMP.jl grows to support practical, industrial applications of inference in sophisticated Bayesian agents and hopefully will also drive more research in this area.

In Section 2, we motivate the need for RMP by analyzing some weaknesses of alternative approaches. Section 3 reviews the background knowledge on message passing-based inference on factor graphs and variational Bayesian inference as a constrained Bethe free energy optimization problem. The rest of the paper discusses our main contributions:

(i) In Section 4, we present the idea of reactive message passing as a way to perform event-driven reactive Bayesian inference by message passing in factor graphs. RMP is a surprisingly simple idea of combining two well-studied approaches from different fields: message passing-based Bayesian inference and reactive programming.

(ii) In Section 5, we present an efficient and scalable implementation of RMP for automated Bayesian inference in the form of the ReactiveMP.jl package for the Julia programming language. We also introduce a specification language for the probabilistic model and inference constraints;

(iii) In Section 6, we benchmark the ReactiveMP.jl on various standard probabilistic signal processing models and compare it with existing message passing-based and sampling-based Bayesian inference implementations. We show that our new implementation scales easily for selected models that include hundreds of thousands of random variables on a regular MacBook Pro laptop computer. For these models, in terms of execution time, memory consumption, and scalability, the proposed RMP implementation Bayesian inference outperforms the existing solutions by hundreds of orders of magnitude and takes roughly a couple of milliseconds or a couple of minutes depending on the size of a data set and a number of random variables in a model.

Finally, in Section 7, we discuss work-in-progress and potential future research directions.

2. Motivation

Open access to efficient software implementations of strong mathematical or algorithmic ideas often leads to sharply increasing advances in various practical fields. For example, backpropagation in artificial neural networks stems from at least the 1980s, but practical applications have skyrocketed in recent years due to new solutions in hardware and corresponding software implementations such as TensorFlow [1] or PyTorch [2].

However, the application of Bayesian inference for real-world signal processing problems still remains a big challenge. If we consider an autonomous robot that tries to find its way in new terrain, we would want it to reason about its environment in real time as well as to be robust to potential failures in its sensors. Furthermore, the robot preferably has the ability to not only adapt to new observations but also to adjust its internal representation of the current environment in real time. Additionally, the robot will have limited computational capabilities and should be energy-efficient. These issues form a very challenging barrier in the deployment of real-time Bayesian inference-based synthetic agents to real-world problems.

In this paper, our goal is to build a foundation for a new approach to efficient, scalable, and robust Bayesian inference in state-space models and release an open-source toolbox to support the development process. By efficiency, we imply the capability of performing real-time Bayesian inference with a limited computational and energy budget. By scalability, we mean that inference execution is performed to an acceptable accuracy with limited resources even if the model size and number of latent variables are very large. Robustness is also an important feature, by which we mean that if the inference system is deployed in a real-world setting, then it needs to stay continually operational even if part of the system collapses.

We propose a combination of message passing-based Bayesian inference on Forney-style factor graphs and the reactive programming approach, which, to our knowledge, is less well known and new in the message passing literature. Our approach is inspired by the neuroscience community and the *Free Energy Principle* [3] because the brain is a good example of a working system that already realizes robust and real-time Bayesian inference at a large scale for a small energy consumption budget.

2.1. Message Passing

2.1. Message Passing. Generative models for complex real-world signals such as speech or video streams are often described by highly factorized probabilistic models with sparse structure and few dependencies among latent variables. Bayesian inference in such models can be performed efficiently by message passing on the edges of factor graphs. A factor graph visualizes a factorized representation of a probabilistic model where edges represent latent variables, and nodes represent functional dependencies among these variables. Generally, as the models scale up to include more latent variables, the fraction of direct dependencies among latent variables decreases, and as a result, the factor graph becomes sparser. For highly factorized models, efficient inference can be realized by message passing, as it naturally takes advantage of the conditional independencies among variables.

State-of-the-art Bayesian inference algorithms based on message passing are traditionally designed with the notion of a globally fixed message passing schedule [4]. The presence of a fixed message passing update schedule comes with a number of disadvantages relative to a reactive system architecture. We rehearse a few issues as follows:

(i) Unpredictable or different update rates in multiple sensor data streams. Consider a model that expects streaming data from two different sensors that may produce samples at different update rates or with unpredictable delays. In this case, a consistent and robust implementation of a fixed message passing schedule is not suitable.

(ii) Robust operability. In many signal processing applications, we would like to be robust against missing data from a failing sensor. Technically, a failing sensor implies a model structure update that would require a temporary system reset to recompute an appropriate global message passing schedule. A reactive system does not need this reset since it does not rely on a message passing schedule that is tightly linked to the model structure.

(iii) Lazy computations. A fixed global schedule does not support executing operations "lazily" and on-demand in different parts of the graph as soon as data arrive. Instead, when the external world changes, the message passing schedule should adapt accordingly. A reactive system does not rely on a schedule and automatically only computes message updates when they are needed.

(iv) Computational complexity. Building an efficient fixed schedule for a large graph with conditional loops is an extremely hard problem that consumes a significant amount of computer resources because it requires a full traversal of the model's graph.

(v) Which schedule is optimal? There are a number of papers that debate the merits of various strategies for building message passing schedules is better [5–7]. However, the problem lies in fixing the schedule per se. The world may change and is always to some degree unpredictable. Since the objective of backward message passing is to absorb (squeeze) prediction errors, a proper schedule cannot be fixed a priori.

While specialized fixed schedule schemes can be highly beneficial in some circumstances [8], in the current paper, we investigate a different implementation paradigm, which does not require explicit scheduling and comes with a number of additional benefits.

2.2. Reactive Programming. In this paper, we provide a fresh look at message passing-based inference from an implementation point of view. We explore the feasibility of using the reactive programming (RP) paradigm [9] as a solution to the problems mentioned above. Essentially, RP supports running computations by dynamically reacting to changes in data sources, thus eliminating the need for a precomputed synchronous message update scheme. The benefits of using RP for different problems have been studied in various fields from physics simulations [10] to neural networks [11] and probabilistic programming as well [12]. We recognize the RP paradigm as the suitable programming abstraction for message passing algorithms and propose a new reactive version of the message passing framework, which we simply call *Reactive Message Passing* (RMP). The new framework is designed to run without any prespecified schedule, autonomously reacts to changes in data, scales to large probabilistic models with hundreds of thousands of unknowns, and, in principle, allows for more advanced features, such as run-time probabilistic model adjustments, parallel inference execution, and built-in support for asynchronous data streams with different update rates.

To support further development, we present our own implementation of the RMP framework in the form of a software package for the Julia programming language called ReactiveMP.jl. We show empirical results and benchmarks of the new implementation for different probabilistic models including a Gaussian linear dynamical system, a hidden Markov model, and a non-conjugate hierarchical Gaussian filter model. More examples, including regression models, mixture models, autoregressive models, normalizing flow models, real-time processing, update rules based on the expectation propagation algorithm, and others are available in the ReactiveMP.jl repository on GitHub. In addition, our implementation has already been tested in battle on sophisticated large-scale models with real-world data sets [13–16].

3. Background

This section first briefly introduces message passing-based exact Bayesian inference on Forney-style factor graphs. Then, we extend the scope to approximate Bayesian inference by variational message passing based on the minimization of the constrained Bethe free energy.

3.1. Forney-Style Factor Graphs. In our work, we use Forney-style factor graphs (FFG) to represent and visualize conditional dependencies in probabilistic models [17–19], but the concept of RMP should be compatible with all other graph-based model representations.

An FFG is an undirected graph with nodes $a \in \mathcal{V}$ and edges $i \in \mathcal{E}$ that can be used to represent a factorized function

$$p(s) = \prod_{a \in \mathcal{V}} p_a(s_a), \tag{1}$$

where p_a are positive functions, and $s_a \in s$ are sets of argument variables for each p_a. Each node a is associated with a corresponding factor p_a, and each edge i is associated with one and only one variable $s_i \in s$. An edge i is connected to a node a if and only if $s_i \in s_a$. We use (see Figure 1) square boxes with letters f, g, and h to represent factors, together with the edges and associated letters x, z, s_i, y etc. to represent variables.

Throughout the paper, we find it convenient to work with *Terminated* FFGs (TFFG) that do not contain half-edges; i.e., each edge is connected to two nodes. This restriction does not affect the class of representable functions since each dangling edge s_i in an FFG can be terminated by a factor $p(s_i) = 1$ without changing the global function. In all of our examples, we assume a TFFG representation unless stated otherwise.

3.2. Inference by Message Passing. We consider a probabilistic model with probability density function (pdf) $p(s, y) = p(y \mid s)p(s)$ where y represents observations and s represents latent variables. The general goal of Bayesian inference is to estimate the *posterior* probability distribution $p(s \mid y = \hat{y})$. Often, it is also useful to estimate *marginal* posterior distributions

$$p\left(s_i \mid y = \hat{y}\right) = \int p(s \mid y = \hat{y}) \mathrm{d}s_{\setminus i}, \tag{2}$$

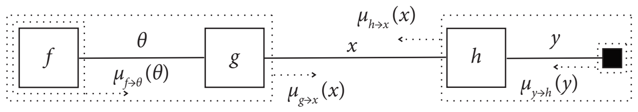


FIGURE 1: The message passing scheme for (3). The factors f, g, and h are represented by nodes and the variables θ, x, and y are associated with edges. The small black node indicates that y has been observed and the posterior for y is therefore clamped to its observed value $\hat{y}$. Technically, one can interpret the small black node as an additional factor $\delta(y-\hat{y})$ that extends the original model. Each nested integral in (3) can be interpreted as a message that flows between nodes on the TFFG.

for the individual components s_i of s. In the case of discrete states or parameters, the probability density function is replaced by a probability mass function (pmf), and the integration is replaced by a summation.

In high-dimensional spaces, the computation of (2) quickly becomes intractable due to an exponential explosion of the size of the latent space [20]. The TFFG framework provides a convenient solution for this problem. As a simple example, consider the factorized distribution

$$p(x,\theta \mid y=\hat{y}) \propto f(\theta)\cdot g(x,\theta)\cdot h(y,x)\cdot \delta(y-\hat{y}), \qquad (3)$$

where f, g, and h are positive functions, x and θ are latent variables, $y=\hat{y}$ is observed, and $\delta(\cdot)$ denotes either a Kronecker delta or a Dirac delta, depending on the context. The underlying factorization of p allows the application of the algebraic distributive law, which simplifies the computation of (2) to a nested set of lower-dimensional integrals, as shown in (4), effectively reducing the exponential complexity of the computation to linear. This model can be represented by a TFFG shown in Figure 1, and each nested integral can be interpreted as a message that flows on the edges among nodes.

$$\begin{aligned} p(x\mid y=\hat{y}) &\propto \int\int p(x,y,\theta)\cdot\delta(y-\hat{y})\mathrm{d}y\mathrm{d}\theta = \\ &= \overbrace{\int \underbrace{f(\theta)}_{\mu_{f\longrightarrow\theta}(\theta)} g(x,\theta)\mathrm{d}\theta}^{\mu_{g\longrightarrow x}(x)} \cdot \underbrace{\int h(y,x)\overbrace{\delta(y-\hat{y})}^{\mu_{y\longrightarrow h}(y)}\,\mathrm{d}y}_{\mu_{h\longrightarrow x}(x)}. \end{aligned} \qquad (4)$$

The resulting marginal $p(x\mid y=\hat{y})$ in (4) is simply equal to the product of two incoming (or colliding) messages on the same edge, divided by the normalization constant

$$p(x\mid y=\hat{y}) = \frac{\mu_{g\longrightarrow x}(x)\cdot\mu_{h\longrightarrow x}(x)}{\int \mu_{g\longrightarrow x}(x)\cdot\mu_{h\longrightarrow x}(x)\mathrm{d}x}. \qquad (5)$$

This procedure for computing a posterior distribution is known as the *Belief Propagation* (BP) or *Sum-Product* (SP) message passing algorithm, and, generally, it requires only the evaluation of low-dimensional integrals over local variables. In some situations, for example, when all messages $\mu_{x_i\longrightarrow f}(x_i)$ have the form of a Gaussian distribution and the factors are linear, it is possible to use closed-form analytical solutions for the messages. These closed-form formulas are also known as *message update rules*.

3.3. Variational Bayesian Inference.

3.3. Variational Bayesian Inference. Inference problems in practical models often involve computing messages in (5) that are difficult to evaluate analytically. In these cases, we may resort to approximate Bayesian inference solutions with the help of variational Bayesian inference methods [21–23]. In general, the variational inference procedure introduces a *variational* posterior $q(s)\in\mathcal{Q}$ that acts as an approximate distribution to the Bayesian posterior $p(s\mid y=\hat{y})$. The set $\mathcal{Q}$ is called a *variational family of distributions*. Typically, in variational Bayesian inference methods, the aim is to minimize the *variational free energy* (VFE) functional

$$F[q] \triangleq \underbrace{\int q(s)\log\frac{q(s)}{p(s\mid y=\hat{y})}\mathrm{d}s}_{\mathrm{KL}[q(s)\|p(s\mid y=\hat{y})]} - \log p(y=\hat{y}), \qquad (6)$$

$$q^*(s) = \underset{q(s)\in\mathcal{Q}}{\operatorname{argmin}}\, F[q]. \qquad (7)$$

In most cases, additional constraints on the set $\mathcal{Q}$ make the optimization task (7) computationally more efficient than the belief propagation but only give an approximate solution for (2). The literature distinguishes two major types of constraints on $\mathcal{Q}$: form constraints and factorization constraints [24]. Form constraints force the approximate posterior $q(s)$ or its marginals $q_i(s_i)$ to be of a specific parametric functional form, for example, a Gaussian distribution $q(s)=\mathcal{N}(s\mid\mu,\Sigma)$. Factorization constraints in the posterior $q(s)$ introduce additional conditional independence assumptions that are not present in the generative model $p(y,s)$.

3.4. Constrained Bethe Free Energy Minimization. The VFE optimization procedure in (7) can be framed as a message passing algorithm, leading to the so-called *Variational message passing* (VMP) algorithm. A very large set of message passing algorithms, including BP and VMP, can also be interpreted as VFE minimization augmented with the *Bethe factorization assumption*

$$q(s) = \prod_{a\in\mathcal{V}} q_a(s_a)\prod_{i\in\mathcal{E}} q_i(s_i)^{-1}, \qquad (8a)$$

$$\int q_i(s_i)\mathrm{d}s_i = 1, \quad \forall i\in\mathcal{E}, \qquad (8b)$$

$$\int q_a(s_a)\mathrm{d}s_a = 1, \quad \forall a\in\mathcal{V}, \qquad (8c)$$

$$\int q_a(s_a)\mathrm{d}s_{a\setminus i} = q_i(s_i), \quad \forall a\in\mathcal{V}, \forall i\in a, \qquad (8d)$$

on the set $\mathcal{Q}$ [25]. In (8a), $\mathcal{E}$ is a set of variables in a model, $\mathcal{V}$ is a set of factors f_a in the corresponding TFFG, (s_a, y_a) is a set of (latent and observed, respectively) variables connected to the corresponding node of f_a, $q_a(s_a)$ refers to a variational posterior for factor $f_a(s_a,y_a)$, and $q_i(s_i)$ is a variational posterior for marginal $p(s_i\mid y=\hat{y})$.

We refer to the Bethe-constrained variational family $\mathcal{Q}$ as $\mathcal{Q}_B$. The VFE optimization procedure in (8a) with Bethe assumption of (8a)–(8d) and extra factorization and form constraints for the local variational distributions $q_a(s_a)$ and $q_i(s_i)$ is called the *constrained Bethe free energy* (CBFE) optimization procedure. Many of the well-known message passing algorithms, including belief propagation, message passing-based expectation maximization, structured VMP, and others, can be framed as CBFE minimization tasks [26, 27]. Moreover, different message passing algorithms in a TFFG can be straightforwardly combined [24], leading to a very large set of possible *hybrid* message passing-based algorithms. All these hybrid variants still allow for a proper and consistent interpretation by performing message passing-based inference through CBFE minimization. Thus, the CBFE minimization framework provides a principled way to derive new algorithms within message passing-based algorithms and provides a solid foundation for a reactive message passing implementation.

4. Reactive Message Passing

This section provides an introduction to reactive programming and describes the core ideas of the reactive message passing framework [9, 28]. First, we discuss the reactive programming approach as a standalone programming paradigm without its relation to Bayesian inference (Section 4.1). Then, we discuss how to connect message passing-based Bayesian inference with reactive programming (Section 4.2).

4.1. Reactive Programming.

This section briefly describes the essential concepts of the reactive programming (RP) paradigm, such as observables, subscriptions, actors, subjects, and operators. Generally speaking, RP provides a set of guidelines and ideas to simplify the application of asynchronous streams of data and/or events.

4.1.1. Observables.

The core idea of the reactive programming paradigm is to replace variables in the context of programming languages with observables. Observables are *lazy push collections*, while regular data structures, such as arrays, lists, or dictionaries, are *pull* collections. The term pull refers here to the fact that a user can directly ask for the state of these data structures or "pull" their values. Observables, in contrast, *push* or *emit* their values over time, and it is not possible to directly ask for the state of an observable but only to *subscribe* to future updates. The term *lazy* means that an observable does not produce any data and does not consume computing resources if no one has subscribed to its updates. Figure 2 shows a standard visual representation of observables in the reactive programming context. We denote an observable of some variable x as $\mathring{x}$. If an observable emits functions (for example, a probability density function) rather than just primitive values such as floats or integers, we write such an observable as $\mathring{f}(x)$. This notation does not imply that the observable is a function of x, but rather than it emits functions of x.

4.1.2. Subscriptions.

To start listening to new updates from some observable, RP uses subscriptions. A subscription (Figure 3(a)) represents the execution of an observable. Each subscription creates an independent observable execution and consumes computing resources.

4.1.3. Actors.

An actor (Figure 3(b)) is a special computational unit that subscribes to an observable and performs some *actions* whenever it receives a new update. In the literature, actors are also referred to as *subscribers* or *listeners*.

4.1.4. Subject.

A subject is a special type of actor that receives an update and simultaneously re-emits the same update to multiple actors. In other words, a subject utilizes a single subscription to some observable and replicates updates from that observable to multiple subscribers. This process of re-emitting all incoming updates is called *multicasting*. One of the goals of a subject is to share the same observable execution and, therefore, save computer resources. A subject is effectively an actor and an observable at the same time.

4.1.5. Operators.

An operator is a function that takes one or more observables as its input and returns another observable. An operator is always a pure operation in the sense that the input observables remain unmodified. Next, we discuss a few operator examples that are essential for the RMP framework.

The first essential operator is the map operator that applies a given *mapping function* to each value emitted by a source observable and returns a new modified observable (Figure 4(a)). When we apply the map operator with a mapping function $\mathscr{M}$ to an observable, we say that this observable is *mapped* by a function $\mathscr{M}$, see Listing 1.

The second essential operator is the combineLatest operator that combines multiple observables to create a new observable whose values are calculated from the latest values of each of the original input observables (Figure 4(b)). We show an example of combineLatest operator application in Listing 2. The combineLatest operator has different update strategies that specify when a resulting observable should emit new updates. For example, emit a new update only after all inner observables have been updated with a new value or emit a new value any time a single inner observable has been updated and reuse previous cached values for the remaining inner observables. The combineLatest operator stores a snapshot only of the latest values in all inner observables and does not store subsequent previous updates in case one of the inner observables is delayed (unbounded buffering is possible with the zip operator instead).

The reactive programming paradigm has many small and basic operators, for example, filter, count, start_with, but the map and combineLatest operators form the foundation for the RMP framework.

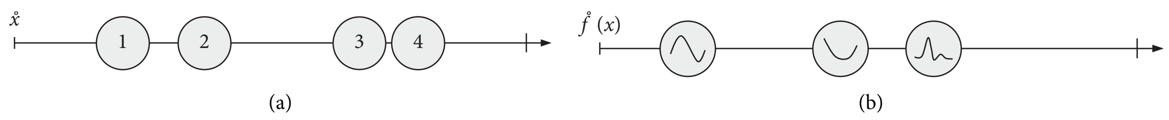


Figure 2: A visual representation of an observable collection over time. An arrow represents a timeline. Circles denote updates at a specific point on that timeline. Values inside circles denote the corresponding data of the update. The bar at the end of the timeline indicates a completion event after which the observable stops sending new updates. (a) An observable emitting primitive (integer) values. (b) An observable emitting functions of x.

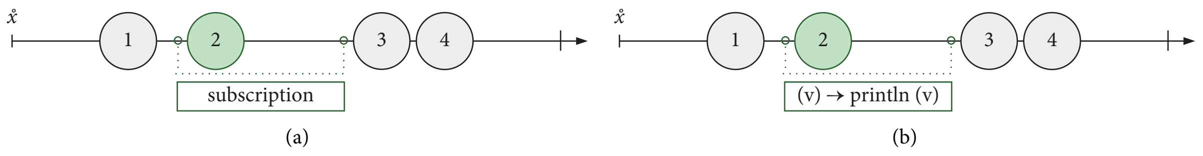
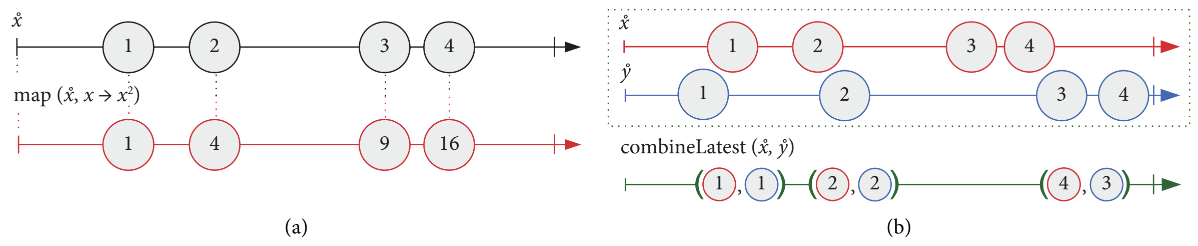


Figure 3: A visual representation of a subscription execution and a simple actor. An arrow represents a timeline. Circles denote updates at a specific point on a timeline. Values inside circles denote the corresponding data of the update. The bar at the end of the timeline indicates a completion event after which the observable stops sending new updates. (a) The subscription happens at a specific point in time and allows actors to receive new values. In this example, an actor would receive only update ②, since the subscription was executed after update ① but was cancelled before updates ③, ④. (b) An actor subscribes to an observable, listens to its updates, reacts, and performs some actions. In this example, the actor simply prints one single value it receives and unsubscribes.



$\mathring{x}$
map ($\mathring{x}$, $x \to x^2$)
1
4
9
16
(a)
$\mathring{y}$
combineLatest ($\mathring{x}$, $\mathring{y}$)
(1, 1)
(2, 2)
(4, 3)
(b)

Figure 4: A visual representation of the reactive operators' application. All reactive operators do not change the original observables. It is still possible to subscribe to the original observables and observe their values over time. (a) The map operator creates a new observable that mirrors the original observable but with transformed values using the provided mapping function. (b) The combineLatest operator combines two or more source observables into a single one and emits a combination of the latest values of all inner source observables.

```
# We create the mapped observable of squared values
# of the original "source" observable
source = get_source ()
squared_source = map (source, x −> x ^ 2)
```

Listing 1: An example of the map operator application. We apply the map operator with $\mathcal{M}(x) = x^2$ to obtain a new observable that emits squared values from the original observable. The original observable remains unmodified.

```
# We assume that "source1" and "source2" are both observables of integers
source1 = get_source1 ()
source2 = get_source2 ()
# We create a new observable by applying a combineLatest operator to it
combined = combineLatest (source1, source2)
# We can go further and apply a "map" operator to the combined observable
# and create a stream of the sum of squares of the latest values
# from "source1" and "source2"
combined_sum = map (combined, (x1, x2) −> x1ˆ2 + x2ˆ2)
```

Listing 2: An example of the combineLatest operator application. We use the combineLatest operator to combine the latest values from two integer streams and additionally apply the map operator with $\mathcal{M}(x_1, x_2) = x_1^2 + x_2^2$. The resulting observable emits the sum of squared values of x_1 and x_2 as soon as both of them emit a new value.

4.2. The Reactive Message Passing Framework. In the next section, we show how to formulate message passing-based inference in the reactive programming paradigm. First, we interpret all variables of interest as observables that change in value over time. These include messages $\mu_{f_j \longrightarrow x_i}(x_i)$ and $\mu_{x_i \longrightarrow f_j}(x_i)$, marginals $q(s_i)$ over all model variables, and local variational posteriors $q_a(s_a)$ for all factors in the corresponding TFFG of a probabilistic generative model. Next, we describe nodes and edges as special types of subjects that react on updates in the message and marginal observables, multicast these updates with the combineLatest operator, and compute the corresponding integrals with the map operator. We use the subscription mechanism to start listening for new updates from marginals in a model. The resulting reactive system resolves message (passing) updates locally, automatically reacts to flowing messages, and updates itself accordingly when new data arrive.

4.2.1. Factor Node Updates. In the context of TFFG, a node consists of a set of connected edges, where each edge refers to a variable in a probabilistic model. The main purpose of a node in the reactive message passing framework is to accumulate updates from all connected edges in the form of message observables $\mathring{\mu}_{s_i \longrightarrow f}(s_i)$ and marginal observables $\mathring{q}(s_i)$ with the combineLatest operator, followed by computing the corresponding outbound messages with the map operator. We refer to a combination of the combineLatest and map operators as the *default computational pipeline*.

To support different message passing algorithms, each node handles a special object that we refer to as a *node context*. The node context essentially consists of the local factorization, the outbound messages form constraints, and optional modifications to the default computational pipeline. In this setting, the reactive node decides, depending on the factorization constraints, *what* to react on and, depending on the form constraints, *how* to react. As a simple example, consider the belief propagation algorithm with the following message computation rule:

$$\mu_{z \longrightarrow h}(z) = \int\int \mu_{x \longrightarrow f}(x)\mu_{y \longrightarrow f}(y) f(x, y, z)\mathrm{d}x\mathrm{d}y, \quad (9)$$

where f is a factor with three arguments x, y, z, $\mu_{z \longrightarrow h}(z)$ is an outbound message on edge z towards node h, and $\mu_{x \longrightarrow f}(x)$ and $\mu_{y \longrightarrow f}(y)$ are inbound messages on edges x and y, respectively. This equation does not depend on the marginals $q(x)$ and $q(y)$ on the corresponding edges; hence, combineLatest needs to take into account only the inbound message observables (Figure 5(a)). In contrast, the variational outbound message update rule for the same node with mean-field factorization assumption

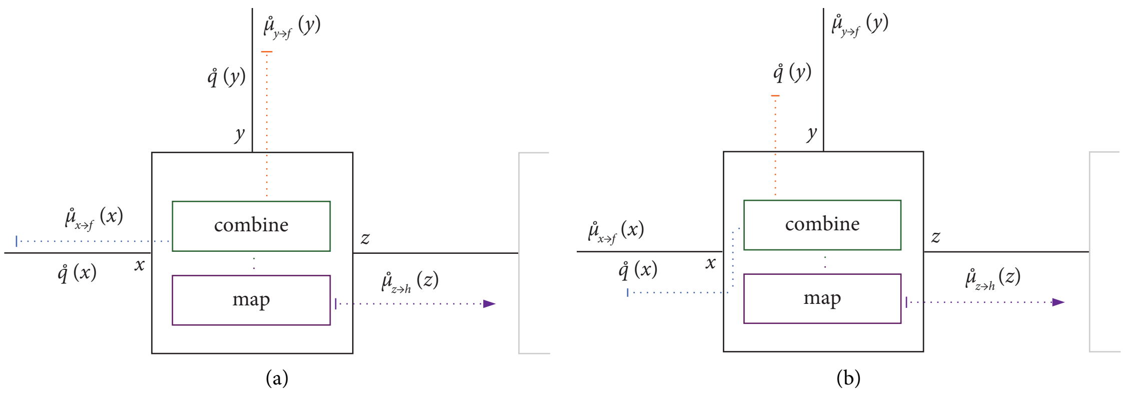
$$\mu_{z \longrightarrow h}(z) = \exp \int\int q(x)q(y)\log f(x, y, z)\mathrm{d}x\mathrm{d}y, \quad (10)$$

does not depend on the inbound messages but rather depends only on the corresponding marginals (Figure 5(b)). It is possible to adjust the node context and customize the input arguments of the combineLatest operator so that the outbound message on some particular edge will depend on any subset of local inbound messages, local marginals, and local joint marginals over a subset of connected variables. From a theoretical point of view, each distinct combination potentially leads to a new type of message passing algorithm [24] and may have different performance characteristics that are beyond the scope of this paper. The main idea is to support as many message passing-based algorithms as possible by employing simple reactive dependencies between local observables. The context also specifies when to react to new updates from local dependencies, namely, either when all dependencies have been updated or when any of the local dependencies have been updated.

After the local context object has been set, a node reacts to new updates in inbound messages or marginals and computes the corresponding outbound messages for each connected edge independently, see Listing 3.

In addition, outbound message observables naturally support custom operators before or after the map operator. The strength of custom operators lies in their ability to apply additional computational steps locally when needed, and these computational steps may be different for different factor nodes. This allows users to select local approximation techniques that match specific scenarios, such as the need to prioritize speed over accuracy or vice versa. Adding a custom operator to the default message computational pipeline may change the corresponding inference algorithm and may help achieve even better performance for some custom models [27, 29]. As an example, one of such custom operators could be the application of the Laplace approximation in cases where an exact analytical message update rule is not available, see Figure 6(a) and Listing 4. It is worth noting that any extra approximation steps, such as the Laplace approximation, will inevitably affect the inference results, and their performance will be model dependent [30]. Other examples of these custom operators may include logging or debugging messages, see Figure 6(b).

4.2.2. Marginal Updates. In the context of TTFG, each edge refers to a single variable s_i. The main purpose of an edge in the reactive message passing framework is to react to outbound messages from connected nodes and emit the corresponding marginal $q(s_i)$. Therefore, we define the marginal observable $\mathring{q}(s_i)$ as a combination of two adjacent outbound message observables from connected nodes with the combineLatest operator and their corresponding normalized product with the map operator, see Listing 5 and Figure 7. Regarding node updates, we refer to the combination of combineLatest and map operators as the default computational pipeline. Each edge has its own *edge context* object that is used to customize the default pipeline. Extra pipeline stages are mostly needed to assign additional form constraints to variables in a probabilistic model, as discussed in Section 3. For example, in the case where no analytical closed-form solution is available for a normalized product of two colliding messages, it is possible to modify the default computational pipeline and to fall back to an available approximation method, such as the point-mass form



$\mathring{\mu}_{y \to f}(y)$
$\mathring{q}(y)$
y
$\mathring{\mu}_{x \to f}(x)$
$\mathring{q}(x)$
x
combine
map
z
$\mathring{\mu}_{z \to h}(z)$
(a)
(b)

FIGURE 5: Graphical representation of the default computational pipeline for different local factorization constraints. Different combination of local input observable sources potentially specifies different message passing algorithms. (a) No extra factorisation assumption for local variational distribution q corresponds to the belief propagation update rule (9). (b) The mean-field factorisation assumption for the local variational distribution q corresponds to the variational message passing update rule (10).

```
context = getcontext (factornode)
# iterate over edges connected to a specific "factornode"
for edge in local_edges (factornode)
    # Combine all updates from a node's local dependencies.
    # This usually includes messages and marginals updates from other edges
    # and depends on the local factorization constraints (see CBFE optimization)
    updates = combineLatest (local_dependencies (factornode, edge, context))
    # Create an outbound message observable by applying
    # a map operator with a suitable "compute_message" procedure
    outbound = map (updates, update −> compute_message (update, context))
    . . .
end
```

LISTING 3: Pseudo-code for generating an outbound message for each edge connected to a factor node.

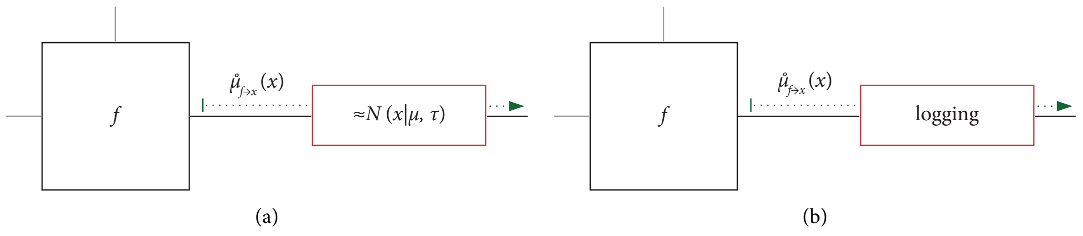


FIGURE 6: A graphical representation of different custom default pipeline modifications for some outbound message observables. The strength of custom operators lies in their ability to apply additional computational steps locally when needed, and these computational steps may be different for different factor nodes. This allows users to select local approximation techniques that match specific scenarios, such as the need to prioritize speed over accuracy or vice versa. (a) An example of applying a Laplace approximation operator to an observable of outbound messages. (b) An example of applying a logging operator to an observable of outbound messages.

```
context = getcontext (factornode)
for edge in local_edges (factornode)
    . . .
    outbound = map (updates, update −> compute_message (update, context))
    # Here we apply a custom approximation operator
    outbound = map (outbound, message −> approximate_as_gaussian (message))
    # and (if needed) logging statements
    outbound = map (outbound, message −> log_into_file (message))
    . . .
end
```

LISTING 4: Pseudo-code for customizing default outbound messages computational pipeline with custom pipeline stages.

```
# iterate over all edges in the entire factor graph
for edge in all_edges (model)
    context = getcontext (edge)
    left_message = get_left_message_updates (edge, context)
    right_message = get_right_message_updates (edge, context)
    # Combine all updates from left and right observables
    updates = combineLatest (left_message, right_message)
    # a map operator with a "prod_and_normalise" procedure
    posterior = map (updates, update -> prod_and_normalise (update, context))
end
```

Listing 5: Pseudo-code for generating an posterior marginal update for each edge in a probabilistic model.

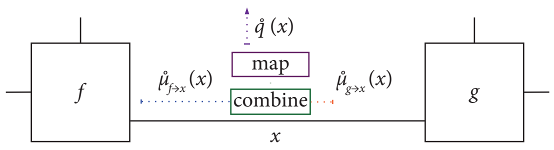
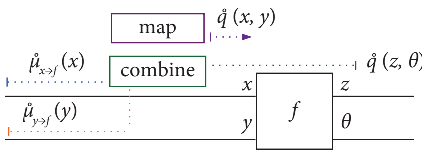


Figure 7: A visual representation of marginal observable computation for some variable x and its corresponding edge with the default computational pipeline. The marginal distribution for a variable is equal to the product of two colliding messages on the corresponding edge divided by a normalisation constant.



map
$\mathring{q}(x, y)$
$\mathring{\mu}_{x\to f}(x)$
combine
$\mathring{q}(z, \theta)$
$\mathring{\mu}_{y\to f}(y)$
x
y
f
z
θ

Figure 8: Visual representation of the local variational distribution observable computation on a subset of edges around a node f with four edges x, y, z, and θ. The local variational distribution for a subset of connected edges depends on inbound messages on these edges and the local variational distribution over remaining edges and corresponding variables.

constraint that leads to the expectation maximization algorithm, or Laplace approximation. The context also specifies when to react to new updates from messages, namely, when all messages have been updated or when any of the messages have been updated.

4.2.3. Joint Marginal Updates. The procedure to compute a local marginal distribution for a subset of connected variables around a node in TFFG is similar to previous cases and has the combineLatest and map operators in its default computational pipeline. To compute a local marginal distribution $q(s_{a\setminus b})$, we need local inbound messages updates to the node f from edges in $s_{a\setminus b}$ and local joint marginal $q(s_b)$. In Figure 8, we show an example of a local joint marginal observable where $s_a = \{x, y, z, \theta\}$, $s_{a\setminus b} = \{x, y\}$, and $s_b = \{z, \theta\}$.

4.2.4. Bethe Free Energy Updates. The CBFE is usually a good approximation to the VFE [24], which is an upper bound to Bayesian model evidence, which in turn scores the performance of a model. Therefore, it is often useful to compute CBFE. To do so, we first decompose (6) into a sum of node-local energy terms $U[f_a, q_a]$ and a sum of local variable entropy terms $H[q_i]$:

$$U[f_a, q_a] = \int q_a(s_a)\log\frac{q_a(s_a)}{f_a(s_a)}\mathrm{d}s_a, \tag{11a}$$

$$H[q_i] = \int q_i(s_i)\log\frac{1}{q_i(s_i)}\mathrm{d}s_i, \tag{11b}$$

$$F[q] = \sum_{a\in\mathcal{V}} U[f_a, q_a] + \sum_{i\in\mathcal{E}} H[q_i]. \tag{11c}$$

We then use the combineLatest operator to combine the local joint marginal observables and marginal observables for each individual variable in a probabilistic model, followed by using the map operator to compute the average energy and entropy terms, see Figure 9 and Listing 6. Each combination emits an array of numbers as an update, and we use the map operator again to compute the sum of such updates.

This procedure assumes that we are able to compute average energy terms for any local joint marginal around any node and to compute entropies of all resulting variable marginals in a probabilistic model. The resulting BFE observable from Listing 6 autonomously reacts on new updates in the probabilistic model and emits a new value as soon as we have new updates for local variational distributions $q_a(s_a)$ and local marginals $q_i(s_i)$.

4.2.5. Infinite Reaction Chain Processing. Procedures from the previous sections create observables that may depend on themselves, especially in the case of a loopy TFFG. This potentially creates an infinite reaction chain. However, the RMP framework naturally handles such cases and infinite data stream processing in its core design, as it uses reactive programming as its foundation. In general, the RP does not make any assumptions about the underlying nature of the update-generating process of observables. Furthermore, it is possible to create a recursive chain of observables where new

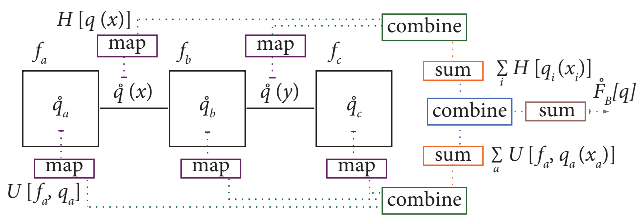


Figure 9: A visual representation of the Bethe free energy observable computation for an arbitrary TFFG with nodes f_a, f_b, and f_c, together with random variables x and y. For each node, the algorithm reacts on observables of variational distributions $\mathring{q}_a$, $\mathring{q}_b$ and $\mathring{q}_c$ and computes corresponding average energy terms with the map operator. For each edge, the algorithm reacts on observables of marginal distributions $\mathring{q}(x)$ and $\mathring{q}(y)$ and computes corresponding entropy terms with the map operator. The resulting updates are combined and transformed with the combineLatest and the sum operators, respectively.

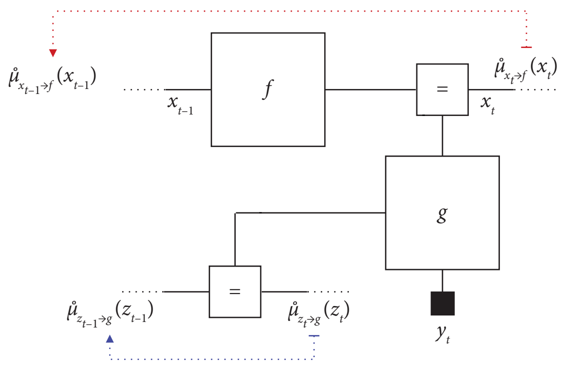
```
# U_local is an array of node–local average energies observables
U_local = [ map (local_q (node), (q_a) –> energy (node, q_a)) for node in nodes ]
# H_local is an array of edge–local entropies observables
H_local = [ map (local_q (edge), (q_i) –> entropy (q_i)) for edge in edges ]
U = map (combineLatest (U_local), u –> sum (u)) # Total average energy observable
H = map (combineLatest (H_local), h –> sum (h)) # Total entropies observable
# The resulting BFE observable emits a sum of
# node–local average energies and edge–local entropies
bfe = map (combineLatest (U, H), update –> sum (update))
```

Listing 6: Pseudo-code for creating the BFE observable (11) for an arbitrary TFFG.

updates in one observable depend on updates in another or even on updates in the same observable.

Furthermore, in view of graphical probabilistic models for signal processing applications, we may create a single time section of a Markov model and redirect a posterior update observable to a prior observable. This method effectively creates an infinite reaction chain that may be interpreted as a factor graph with messages flowing only in a forward-time direction. We refer to such an architecture as an *infinite factor graph*, because it supports data streams with potentially an infinite number of sequential observations and performs Bayesian inference as soon as the data arrive, see Figure 10. By carefully choosing strategies for the combineLatest operator (see Section 4.1.5), we can avoid an infinite messaging loop, and we will show an example of such an inference application in Section 6.3.

This setting creates an infinite stream of posteriors redirected to a stream of priors. The resulting system reacts to new observations y_t and computes messages automatically as soon as a new data point is available. However, in between new observations, the system stays idle and simply waits. In the case of variational Bayesian inference, we may choose to perform more VMP iterations during the idle time so as to improve the approximation of (2).



$\mathring{\mu}_{x_{t-1}\to f}(x_{t-1})$
x_{t-1}
f
$=$
$\mathring{\mu}_{x_t\to f}(x_t)$
x_t
g
$=$
$\mathring{\mu}_{z_{t-1}\to g}(z_{t-1})$
$\mathring{\mu}_{z_t\to g}(z_t)$
y_t

Figure 10: An example of an infinite factor graph in the RMP framework with arbitrary state transition nodes f and g. The small black square indicates the observed variable. The dotted red line indicates a redirection of an observable of posterior updates for x_t to an observable of priors for x_{t-1}. The dotted blue line indicates a redirection of an observable of posterior updates for z_t to an observable of priors for z_{t-1}. The model reacts to new incoming data points y_t and computes messages in a time-forward direction only.

5. Implementation

Based on Sections 2 and 3, a generic toolbox for message passing-based constrained Bethe free energy minimization on TFFG representations of probabilistic models should at least comprise the following features:

(1) A comprehensive specification language for probabilistic models $p(y, s)$ of interest.

(2) A convenient way to specify additional local constraints on $\mathcal{Q}_B$ so as to restrict the search space for posterior $q(s)$. Technically, items 1 and 2 together support the specification of a CBFE optimization procedure.

(3) Provide an automated, efficient, and scalable engine to minimize the CBFE by message passing-based inference.

Implementation of all three features in a user-friendly and efficient manner is a very challenging problem. We have developed a new reactive message passing-based Bayesian inference package, which is fully written in the open-source, high-performant scientific programming language Julia [31]. Our implementation makes use of the Rocket.jl package, which we developed to support the reactive programming in Julia. The Rocket.jl implements observables, subscriptions, actors, and basic reactive operators such as the combineLatest and the map and makes it easier to work with asynchronous data streams. We use the GraphPPL.jl package to simplify the probabilistic model specification and variational family constraints specification parts. We refer to this ecosystem as a whole as **ReactiveMP.jl** (ReactiveMP.jl GitHub repository https://github.com/biaslab/ReactiveMP.jl).

It is worth noting that our implementation is a work in progress and is continually improving to support a wider range of probability distributions and nonconjugate relationships among variables. The implementation (at the time of submission of this paper) aims to support a large selection of often-used members from the exponential family of distributions. The public API of the package makes it easy for users to add custom factor nodes and custom message update rules for missing distributions or deterministic functions. We plan to maintain the ecosystem in the future, support more distributions, and improve the general performance of the code.

5.1. Model Specification. ReactiveMP.jl uses a @model macro for both the model and constraints specification. The @model macro returns a function that generates the model with specified constraints. It has been designed to resemble existing probabilistic model specification libraries in Julia but also to be flexible enough to support reactive message passing-based inference as well as factorization and form constraints for variational families $\mathcal{Q}_B$ from (8a). As an example of the model specification syntax, consider the simple model in Listing 7.

The tilde operator (~) creates a new random variable with a given functional relationship with other variables in a model and can be read as "is sampled from…" or "is modeled by…." Sometimes, it is useful to create a random variable in advance or to create a sequence of random variables in a probabilistic model for improved efficiency. The ReactiveMP.jl package exports a randomvar() function to manually create random variables in the model, see Listing 8.

To create observations in a model, the ReactiveMP.jl package exports the datavar(T) function, where T refers to a type of data such as Float64 or Vector{Float64}. The model reacts to these observations and performs inference automatically as soon as new data arrive, see Listing 9.

The tilde operator supports an optional **where** statement to define local node- or variable-context objects. As discussed in Section 4.2, local context objects specify local factorization constraints for the variational family of distributions $\mathcal{Q}_B$ or may include additional pipeline stages to the default computational pipeline of messages and marginals updates, see Listing 10.

5.2. Marginal Updates. The ReactiveMP.jl framework API exports the getmarginal () function, which returns a reference for an observable of marginal updates. Reactivity in the core design makes it especially easy for other parts of an application to react on updated marginals and perform some complex actions or make decisions based on updated marginals in real time. In the Listing 11, we show an example usage of the subscribe! () function to subscribe to future marginal updates and perform a simple action as soon as new update is available.

5.3. Bethe Free Energy Updates. The ReactiveMP.jl framework API exports the score () function to get a stream of Bethe free energy values, see Listing 12. By default, the Bethe free energy observable emits a new update only after all variational posterior distributions $q_a(s_a)$ and $q_i(s_i)$ have been updated.

5.4. Multiple Dispatch. An important problem is how to select the most efficient update rule for an outbound message, given the types of inbound messages and/or marginals. In general, we want to use known analytical solutions if they are available and otherwise select an appropriate approximation method. In the context of software development, the choice of which method to execute when a function is applied is called *dispatch*.

Locality is a central design choice of the reactive message passing approach, but it makes it impossible to infer the correct inbound message types locally before the data have been seen. Fortunately, the Julia language supports *dynamic multiple dispatch*, which allows for an elegant solution to this problem. Julia allows the dispatch process to choose which of a function's methods to call based on the number of arguments given and on the types of all of the function's arguments at run time. This feature enables automatic dispatch of the most suitable outbound message computation rule, given the functional form of a factor node and the functional forms of inbound messages and/or posterior marginals.

Julia's built-in features also support the ReactiveMP.jl implementation to dynamically dispatch to the most efficient message update rule for both exact and approximate variational algorithms. If no closed-form analytical message update rule exists, Julia's multiple dispatch facility provides several options to select an alternative (more computationally demanding) update as discussed in Section 4.2.1.

The ReactiveMP.jl uses the following arguments to dispatch to the most efficient message update rule (see Listing 13):

(i) Functional form of a factor node, for example, Gaussian or Gamma;

(ii) Name of the connected edge, for example, :out;

```
# We use the "@model" macro to accept a Julia function as input.
# In this example, the model accepts a keyword argument 'n' that denotes
# the number of observations in the data set
@model function coin_model (; n)
    # Reactive data inputs for Beta prior over θ random variable
    # so we don't need actual data at model creation time
    a = datavar (Float64)
    b = datavar (Float64)
    # We use the tilde operator to define a probabilistic relationship between
    # random variables and data inputs. It automatically creates a new random
    # variable in the current model and the corresponding factor nodes
    θ ~ Beta (a, b) # creates both "θ" random variable and Beta factor node
    # A sequence of observations with length "n"
    y = datavar (Float64, n)
    # Each observation is modeled by a "Bernoulli" distribution
    # that is governed by a 'θ' parameter
    for i in 1 : n
    y [i] ~ Bernoulli(θ) # Reuses "y [i]" and "θ" and creates "Bernoulli" node
    end
    # "@model" function must have a "return" statement.
    # Later on, the returned references might be useful to obtain the marginal
    # updates on variables in the model and to pass new observations to data
    # inputs so the model can react to changes in data
    # "@model" also modifies the output and always returns the "model"s graph
    return y, a, b, θ
end
```

Listing 7: An example of a probabilistic model specification with the @model macro.

```
x = randomvar () # A single random variable in the model
x = randomvar (n) # A collection of n random variables
```

Listing 8: An example code of random variable creation.

```
y = datavar (Float64) # A single data input of type Float64
y = datavar (Vector{Float64}) # A single data input of type Vector{Float64}
y = datavar (Float64, n) # A sequence of n data inputs of type Float64
```

Listing 9: An example code of data inputs creation.

```
# We can use "where" clause to specify extra factorization
# constraints for the local variational distribution
θ ~ Beta (a, b) where { q = q (θ) q (a) q (b) }
# Structured factorization assumption
x_next ~ NormalMeanVariance (x_previous, tau) where {
q = q (x_next, x_previous) q (tau)
}
# We can also use "where" clause to modify the default
# computational pipeline for all outbound messages from a node
# In this example "LoggerPipelineStage"modifies the default pipeline
# so that it starts to print all outbound messages into standard output
y ~ Gamma (a, b) where { pipeline = LoggerPipelineStage () }
```

Listing 10: An example code for specifying extra factorization constraints for the local variational distribution and the pipeline modifications.

```
# The "coin_model" function returns a reference to the corresponding graph and
# the actual output from the model specification
graph, (y, a, b, θ) = coin_model (; n = 100)
# "subscribe!" accepts an observable as its first argument and
# a callback function as its second argument
θ_subscription = subscribe! (getmarginal (θ), (update) −> println (update))
```

Listing 11: An example code for subscribing on new marginal updates.

```
bfe_updates = score (BetheFreeEnergy (), model)
bfe_subscription = subscribe! (bfe_updates, (update) −> println (update))
```

Listing 12: An example code for BFE updates subscription.

```
# Define a stochastic node of Gaussian type with 3 edges: "out," "mean" and "var"
@node Gaussian Stochastic [ out, mean, var ]
# Define a structured VMP message update rule with 'Marginalisation" constraint
# from "out" edge, which uses:
# – a message on edge "mean"
# – a marginal on edge "var"
@rule Gaussian (:out, Marginalisation) (m_meanGaussian, q_varAny) = begin
    return Gaussian (mean (m_mean), 1/(var (m_mean) + mean (q_var)))
end
```

Listing 13: An example of node and VMP message update rule specification for a univariate Gaussian distribution with mean-precision parametrisation.

(iii) Local variable constraints, for example, Marginalisation or MomentMatching;

(iv) Input messages names and types with m_ prefix, for example, m_mean::Gaussian;

(v) Input marginals names and types with q_prefix, for example, q_var::Any;

(vi) Optional context object, for example, a strategy to correct nonpositive definite matrices, an order of autoregressive model or an optional approximation methods used to compute messages.

All of this together allows the ReactiveMP.jl framework to automatically select the most efficient message update rule. It should be possible to emulate this behavior in other programming languages, but Julia's core support for efficient dynamic multiple dispatch is an important reason why we selected it as the implementation language. It should be noted that the Julia programming language generates efficient and scalable code with run-time performance similar to that of C and C++ [32].

6. Experimental Evaluation

In this section, we provide the experimental results of our RMP implementation on various Bayesian inference problems that are common in signal processing applications. The main purpose of this section is to show that reactive message passing and its ReactiveMP.jl implementation, in particular, are capable of running inference for different common signal processing applications and to explore its performance characteristics.

Each example in this section is self-contained and is aimed at modeling particular properties of the underlying data sets. The linear Gaussian state-space model example in Section 6.1 models a signal with continuously valued latent states and uses a large static data set with hundreds of thousands of observations. In Section 6.2, we present a hidden Markov model for a discretely valued signal with unknown state transition and observation matrices. The hierarchical Gaussian filter example in Section 6.3 shows online learning (filtering) in a hierarchical model with nonconjugate relationships between variables and makes use of a custom factor node with custom approximate message computation rules. It is worth noting that all models presented can be used in both filtering and smoothing settings. Each example in this section has a comprehensive, interactive, and reproducible demo on GitHub (All experiments are available at https://github.com/biaslab/ReactiveMPPaperExperiments) experiments repository. The datasets generated during and/or analysed during the current study are available in the same repository. All experiments have been performed on ReactiveMP of version 1.1.0. More models, tutorials, and advanced usage examples are available in the ReactiveMP.jl package repository on

GitHub (More models, tutorials, and advanced usage examples are available at https://github.com/biaslab/ReactiveMP.jl/tree/v1.1.0/demo). For verification, we used synthetically generated data, but, as we mentioned in Section 2, ReactiveMP.jl has been battle-tested on more sophisticated models with real-world data sets [13–16].

For each experiment, we compared the performance of the new reactive message passing-based inference engine with another message passing package ForneyLab.jl (version 0.11.3, [33]) and the sampling-based inference engine Turing.jl (version 0.19.0, [34]). We selected Turing.jl as a flexible, mature, and convenient platform to run sampling-based inference algorithms in the Julia programming language. In particular, it provides a particle Gibbs sampler for discrete parameters, as well as a Hamiltonian Monte Carlo sampler for continuous parameters. We show that the new reactive message passing-based solution not only scales better but also yields more accurate posterior estimates for the hidden states of the outlined models in comparison with sampling-based methods. Note, however, that Turing.jl is a general-purpose probabilistic programming toolbox and provides more algorithms and instruments to run Bayesian inference in even a broader class of probabilistic models than the current implementation of ReactiveMP.jl.

We do not compare our implementation with other probabilistic programming libraries from other programming languages (e.g., Stan, BUGS, or Pyro) and rather deliberately restrict our comparison only between packages written in the Julia programming languages. The main reason is that the idea of benchmark comparisons in this section is to compare message passing with the state-of-the-art sampling-based methods and measure their performance characteristics on specific models, rather than to compare different packages. In fact, Turing.jl, according to its development team, is almost a direct reimplementation of Stan PPL, has similar performance characteristics, and, for some models, is even more expressive [34].

Variational inference algorithms in general and ReactiveMP.jl in particular use the minimized CBFE value for scoring the model's performance. In contrast, some packages for Bayesian inference methods only compute posterior distributions and ignore the CBFE. To compare the posterior results for the different Bayesian inference methods that do not compute the CBFE functional, we performed a posterior estimation accuracy test by the metric

$$AE[q] = \frac{1}{|\mathscr{D}|} \sum_{d \in \mathscr{D}} \left[\frac{1}{T} \sum_{t=1}^{T} \mathbb{E}_{q(x_t)}[f(x_t - r_t)] \right], \tag{12}$$

where $\mathscr{D}$ is a set of all synthetic datasets d for a particular model, T is a number of time steps used in an experiment, $q(x_t)$ is a resulting posterior $p(x_t \mid y = \hat{y})$ at time step t, r_t is an actual value of the real underlying signal at time step t, f is any positive definite transform. In our experiments, we used $f(x) = x^{'}x$ for continuous multivariate variables, $f(x) = x^2$ for continuous univariate variables and $f(z) = |z|$ for discrete variables. We call this metric the *average error* (AE) metric.

All benchmarks have been performed with the help of the BenchmarkTools.jl package [35], which is a framework to write, run, and compare groups of benchmarks in Julia. For benchmarks, we used the Julia version 1.6, which, at the time the article was written, is the LTS (long-time support) version. All experiments were executed on a MacBookPro-2018 with 2.6 GHz Intel Core-i7 and 16 GB 2400 MHz DDR4 RAM. We show that the new implementation runs smoothly on a regular laptop and does not require either sophisticated supercomputers or GPU support in order to execute Bayesian inference for hundreds of thousands of observations. Therefore, in principle, these experiments can be executed on low-power devices without GPU, such as a Raspberry-Pi.

6.1. Linear Gaussian State-Space Model.

6.1. Linear Gaussian State-Space Model. As our first example, we consider the linear Gaussian state-space model (LG-SSM) that is widely used in the signal processing and control communities [36]. The simple LG-SMM is specified by

$$\begin{aligned} p(x_t \mid x_{t-1}) &= \mathcal{N}(x_t \mid Ax_{t-1}, P), \\ p(y_t \mid x_t) &= \mathcal{N}(y_t \mid Bx_t, Q), \end{aligned} \tag{13}$$

where y_t is the observation at time step t, x_t represents the latent state at time step t, A and B are the state transition and observation model matrices, respectively, and P and Q are Gaussian noise covariance matrices.

6.1.1. Model Specification. Bayesian inference in this type of model can be performed by filtering and smoothing. Although many different variants exist, in our example, we focus on the Rauch–Tung–Striebel (RTS) smoother [37], which can be interpreted as performing the belief propagation algorithm on the full model graph. The belief propagation algorithm implies that we do not impose additional factorization constraints on the variational family of distributions $\mathcal{Q}_B$ and that we need to perform only a single message passing iteration. We show the model specification of this example in Listing 14. Figure 11 shows the comparison of the hidden states recovered between the RMP and HMC methods. Code examples for the Kalman filter inference procedure for this type of model can be found in the RMP experiments repository on GitHub.

6.1.2. Benchmark. The main benchmark results are presented in Figure 12 and Table 1. The accuracy results in terms of the average error metric of (12) are presented in Table 2. The new implementation based on the passing of reactive messages for Bayesian inference shows better results in performance and scalability and outperforms the compared packages in terms of time and memory consumption (Not present in the table, available at https://github.com/biaslab/ReactiveMPPaperExperiments) significantly. Furthermore, the ReactiveMP.jl package is capable of running inferences on very large models with hundreds of thousands of variables. Accuracy results in Table 2 show that the methods based on message passing give more accurate results in

```
@model function linear_gaussian_state_space_model (n, d, A, B, P, Q)
    # We create a sequence of random variables of length "n"
    x = randomvar (n)
    # We create a sequence of observed variables of length "n"
    y = datavar (Vector{Float64}, n)
    # Prior distribution for x [1], "d" is the dimension of observations
    # Here we use mean–covariance parametrisation
    # for Gaussian distribution, but it is also possible to use different
    # parametrisations such as mean–precision or weighted–mean–precision
    x [1] ~ MvGaussianMeanCovariance(zeros (d), 100.0 * diageye (d))
    y [1] ~ MvGaussianMeanCovariance(B * x [1], Q)
    for t in 2:n
        x [t] ~ MvGaussianMeanCovariance(A * x [t−1], P)
        y [t] ~ MvGaussianMeanCovariance(B * x [t], Q)
    end
    # We return "x" and "y" for later reference
    return x, y
end
```

Listing 14: An example of model specification for the linear Gaussian state-space model (14a).

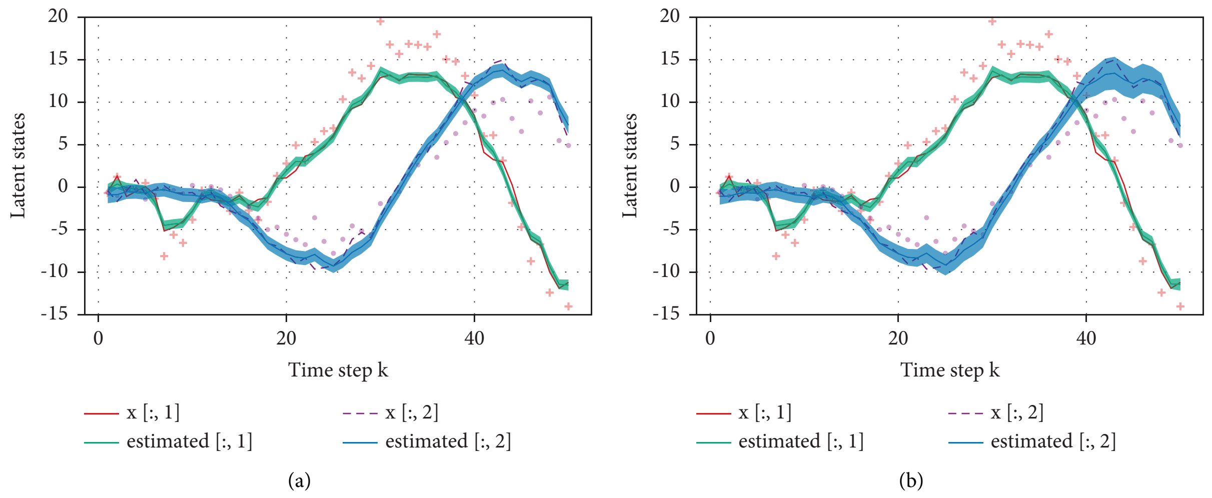


Figure 11: An example of the Bayesian inference results for the linear Gaussian state-space model (12) with multivariate (2-dimensional) observations. The x-axis represents the time steps and the y-axis corresponds to the actual value of hidden states and observations for each dimension. (a) Simulated trajectory, observations, and the inference results using reactive message passing method. (b) Simulated trajectory, observations and the inference results using Hamiltonian Monte Carlo method with 250 samples.

comparison to the sampling methods, which is expected since the method of message passing performs exact Bayesian inference in this type of model.

Figure 12 shows that the ForneyLab.jl package actually executes the inference task for this model faster than the ReactiveMP.jl package. The ForneyLab.jl package thoroughly analyzes the TFFG during precompilation and is able to create an efficient predefined message update schedule ahead of time that is able to execute the inference procedure very fast. Unfortunately, as we discussed in Section 2, ForneyLab.jl's schedule-based solution suffers from long latencies in the graph creation and precompilation stages. This behavior limits investigations of "what-if" scenarios in the model specification space. On the other hand, ReactiveMP.jl creates and executes the reactive message passing scheme dynamically without full graph analysis. This strategy helps with scalability for very large models but comes with some run-time performance costs.

The execution timings of the Turing.jl package and the corresponding HMC algorithm mostly depend on the number of samples in the sampling procedure and other hyperparameters, which usually need to be fine-tuned. In general, in sampling-based packages, a larger number of samples lead to better approximations, but longer run times. On the other hand, as we mentioned at the beginning of Section 6, the Turing.jl platform and its built-in algorithms support running inference in a broader class of probabilistic models, as it does not depend on analytical solutions and is not restricted to work with closed-form update rules.

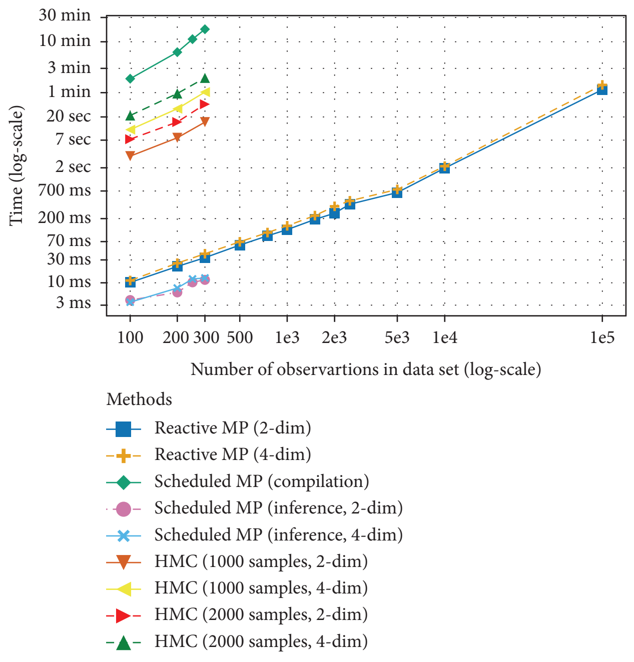


Figure 12: A comparison of run-time duration in milliseconds for automated Bayesian inference a linear Gaussian state-space model (12) across different methods: reactive message passing (ReactiveMP.jl), scheduled message passing (ForneyLab.jl) and Hamiltonian Monte Carlo (Turing.jl), see Table 1. The values in the figure show the minimum possible duration across multiple runs. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl uses HMC sampling with 1000 and 2000 samples respectively. We present benchmark results for more than 1 000 observations only for the ReactiveMP.jl package since for the other compared packages this involved running the inference procedure for more than an hour.

Table 1: A comparison of run-time duration in milliseconds for automated Bayesian inference for a linear Gaussian state-space model (12) across different methods: reactive message passing (ReactiveMP.jl), scheduled message passing (ForneyLab.jl), and Hamiltonian Monte Carlo (Turing.jl). The values in the table show the minimum possible duration across multiple runs. The T column represents the number of observations in a data set, 2-dim and 4-dim columns separate timings for 2-dimensional and 4-dimensional observations respectively. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl uses HMC sampling with 1000 and 2000 samples respectively. We present benchmark results for more than 10 000 observations only for the ReactiveMP.jl package since for other compared packages it involved running the inference for more than an hour.

T	Reactive MP		Scheduled MP			HMC			
			Compilation	Inference		1000 samples		2000 samples	
	2-dim	4-dim		2-dim	4-dim	2-dim	4-dim	2-dim	4-dim
100	10	11	112753	4	4	3389	11246	7302	21337
200	22	26	374830	6	8	7781	28969	15882	57774
300	33	40	1057691	12	13	15822	61575	35545	116967
10000	1971	2152	—	—	—	—	—	—	—
100000	68765	85243	—	—	—	—	—	—	—

Table 2: Comparison of posterior results accuracy in terms of metric (12) in the linear Gaussian state-space model (14a) across different methods: message passing (ReactiveMP.jl and ForneyLab.jl) and Hamiltonian Monte Carlo (Turing.jl). Lower values indicate better performance. ReactiveMP.jl and ForneyLab.jl use message passing-based RTS smoothing algorithms on the full graph. Turing.jl uses HMC sampling with 1000 and 2000 samples, respectively.

	Number of observations					
	2-Dimensional			4-Dimensional		
	100	200	300	100	200	300
Message passing	3.45	3.38	3.30	6.75	6.62	6.58
HMC (1000)	6.21	11.33	26.49	15.17	24.07	42.53
HMC (2000)	4.62	6.24	10.54	10.02	12.76	18.27

We present benchmark results for 10 000 and 100 000 observations only for the ReactiveMP.jl package, since for other compared packages, it involved running the inference for more than an hour. For example, we can estimate that for a static data set with 100 000 observations, the corresponding TFFG for this model has roughly 400 000 nodes. The ReactiveMP.jl executes inference for such a large model in under two minutes. Therefore, we conclude that the new reactive message passing implementation scales better with the number of random variables and is able to run efficient Bayesian inference for large conjugate state-space models.

6.2. Hidden Markov Model. In this example, the goal is to perform Bayesian inference in a hidden Markov model (HMM). An HMM can be viewed as a specific instance of a state-space model in which the latent variables are discretely valued. HMMs are widely used in speech recognition [38, 39], natural language modeling [40], and in many other related fields.

We consider an HMM specified by

$$p(A) = \mathrm{MatrixDirichlet}(A \mid P_A), \tag{14a}$$

$$p(B) = \mathrm{MatrixDirichlet}(B \mid P_B), \tag{14b}$$

$$p(z_t \mid z_{t-1}) = \mathrm{Cat}(z_t \mid A z_{t-1}), \tag{14c}$$

$$p(y_t \mid z_t) = \mathrm{Cat}(y_t \mid B z_t), \tag{14d}$$

where A and B are state transition and observation model matrices, respectively, z_t is a discrete M-dimensional one-hot coded latent state, y_t is the observation at time step t, $\mathrm{MatrixDirichlet}(\cdot \mid P)$ denotes a matrix variate generalization of Dirichlet distribution with concentration parameters matrix P [41], and $\mathrm{Cat}(\cdot \mid p)$ denotes a categorical distribution with concentration parameters vector p. One-hot coding of z_t implies that $z_{t,i} \in \{0, 1\}$ and $\sum_{i=1}^{M} z_{t,i} = 1$. With this encoding scheme, (14d) is short-hand for

$$p(z_{t,i} = 1 \mid z_{t-1,j} = 1) = A_{ij}. \tag{15}$$

6.2.1. CBFE Specification. Exact Bayesian inference for this model is intractable, and we resort to approximate inference by message passing-based minimization of CBFE. For the variational family of distributions $\mathcal{Q}_B$, we assume a structured factorization around state transition probabilities and the mean-field factorization assumption for every other factor in the model

$$q(z, A, B) = q(z) q(A) q(B), \tag{16a}$$

$$q(z) = \frac{\prod_{t=2}^{T} q(z_{t-1}, z_t)}{\prod_{t=2}^{T} q(z_t)}. \tag{16b}$$

We show the model specification code for this model with the extra factorization constraints (16b) specification in Listing 15 and the inference results in Figure 13. Figure 13(b) shows the convergence of the BFE values after several VMP iterations. The qualitative results in Figure 13(a) show a reasonably correct posterior estimation of discretely valued hidden states.

6.2.2. Benchmark. The main benchmark results are presented in Figure 14 and Table 3. Figure 15 presents a comparison of the performance of ReactiveMP.jl as a function of the number of VMP iterations and shows that the resulting inference procedure scales linearly both on the number of observations and the number of VMP iterations performed. In the context of VMP, each iteration decreases the Bethe free and effectively leads to a better approximation for the marginals over the latent variables. As in our previous example, we show the accuracy results for the message passing-based methods in comparison to the sampling-based methods in terms of metric (12) in Table 4. The ForneyLab.jl shows the same level of posterior accuracy, as it uses the same VMP algorithm, but is slower in model compilation and execution times. The Turing.jl is set to use the HMC method for estimating the posterior of transition matrices A and B with a combination of particle Gibbs (PG) for discrete states z.

6.3. Hierarchical Gaussian Filter. For our last example, we consider Bayesian inference in the hierarchical Gaussian filter (HGF) model. The HGF is popular in the computational neuroscience literature and is often used for Bayesian modeling of volatile environments, such as uncertainty in perception or financial data [42]. The HGF is essentially a Gaussian random walk, where the time-varying variance of the random walk is determined by the state of a higher level process, for example, a Gaussian random walk with fixed volatility or another more complex signal. Specifically, a simple HGF is defined as

$$\begin{aligned} p\left(s_t^{(j)} \mid s_{t-1}^{(j)}\right) &= \mathcal{N}\left(s_k^{(j)} \mid s_{t-1}^{(j)}, f\left(s_t^{(j+1)}\right)\right), \\ &\cdot \text{for } j = 1, 2, \ldots, J, \end{aligned} \tag{17a}$$

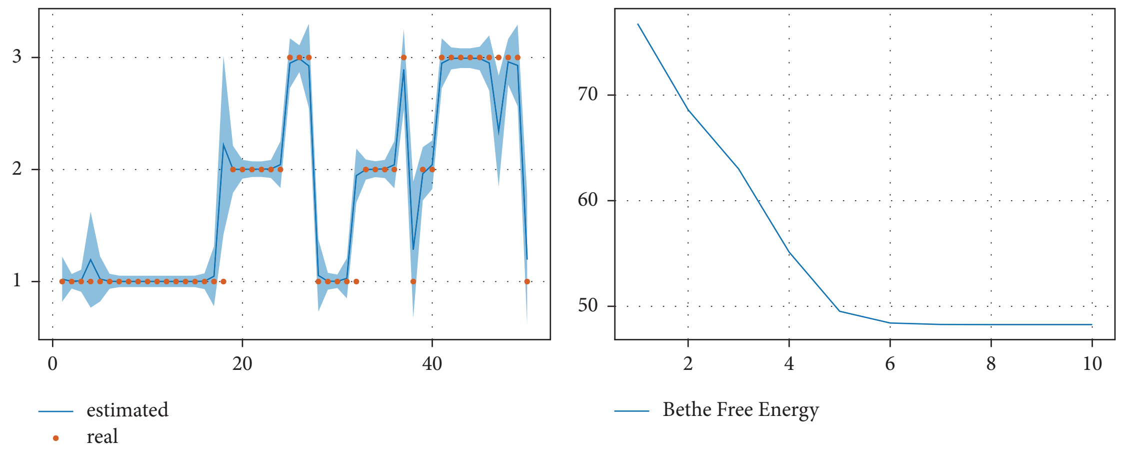
$$p\left(y_t \mid s_t^{(1)}\right) = \mathcal{N}\left(y_t \mid s_t^{(1)}, \tau\right), \tag{17b}$$

where y_t is the observation at time step t, $s_t^{(j)}$ is the latent state for the j-th layer at time step t, and f is a link function, which maps states from the (next higher) $(j + 1) - th$ layer to the nonnegative variance of the random walk in the j-th layer. The HGF model typically defines the link function as $f(s_t^{(j+1)}) = \exp(\kappa s_t^{(j+1)} + \omega)$, where κ and ω are either hyperparameters or random variables included in the model.

As in the previous example, the exact Bayesian inference for this type of model is not tractable. Moreover, the variational message update rules (9) are not tractable either, as the HGF model contains nonconjugate relationships among

```
# "@model" macro accepts an optional list of default parameters
# Here we use the "MeanField" factorization assumption
@model [ default_factorization = MeanField () ]
function hidden_markov_model (n, priorA, priorB)
    A ~ MatrixDirichlet (priorA)
    B ~ MatrixDirichlet (priorB)
    z = randomvar (n)
    y = datavar (Vector{Float64}, n)
    # "Transition" node is an alias for "Categorical (B * z [ ])"
    z [1] ~ Categorical (fill (1.0/3.0, 3))
    y [1] ~ Transition (z [1], B)
for t in 2 : n
    # We override the default "MeanField" assumption with
    # structured posterior factorization assumption using "where" block
    z [t] ~ Transition(z [t − 1], A) where {q = q (z [t − 1], z [t]) q (A)}
    y [t] ~ Transition(z [t], B)
    end
    return A, B, z, y
end
```

Listing 15: An example of model specification for the hidden Markov model (14).



(a) (b)

Figure 13: Inference results for the hidden Markov model in Listing 15. (a) Hidden states inference results. The orange dots represent real values of states at each time step. The blue line represents the mean of the posterior over latent states with one standard deviation. (b) Bethe free energy evaluation results. The x-axis represents an index of VMP iteration. The y-axis represents a Bethe free energy value at a specific VMP iteration.

variables in the form of the link function f. However, inference in this model is still possible with the custom message update rules approximation [43]. The ReactiveMP.jl package supports a straightforward API to add custom nodes with custom message passing update rules. In fact, a large part of the ReactiveMP.jl library is a collection of well-known factor nodes and message update rules implemented using the same API as for custom novel nodes and message update equations. The API also supports message update rules that involve extra nontrivial approximation steps. In this example, we added a custom *Gaussian controlled variance* (GCV) node to model the nonlinear time-varying variance mapping from different hierarchy levels (17a), with a set of approximate update rules based on the Gauss–Hermite cubature rule from [44]. The main purpose of this example is to show that ReactiveMP.jl is capable of running inference in complex nonconjugate models but requires creating custom factor nodes and choosing appropriate integral calculation approximation methods.

6.3.1. CBFE Specification. In this example, we want to show an example of reactive online Bayesian learning (filtering) in a 2-layer HGF model (see Section 4.2.5), but there are no principled limitations to run this model on a full graph. For simplicity and to avoid extra clutter, we assume τ_k, κ, and ω to be fixed, but there are no principled limitations to make them random variables, endow them with priors, and

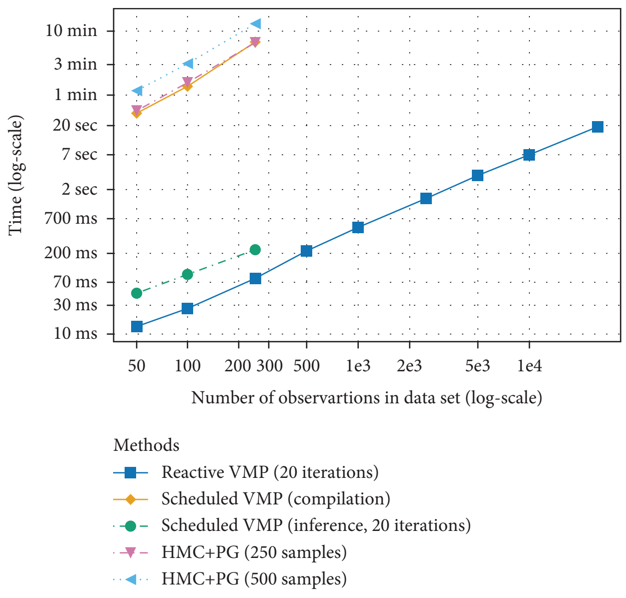


Figure 14: A comparison of run-time duration in milliseconds for automated Bayesian inference for a hidden Markov model (13) across different methods: reactive variational message passing (ReactiveMP.jl), scheduled variational message passing (ForneyLab.jl) and a combination of Hamiltonian Monte Carlo and particle Gibbs (Turing.jl). The values in the figure show the minimum possible duration across multiple runs. In this benchmark the number of categories M of observations is set to 3. ReactiveMP.jl and ForneyLab.jl perform VMP on a full graph. The number of VMP iterations is set to 20. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl runs two benchmarks with 250 and 500 number of samples respectively. We present benchmark results for more than 1 000 observations only for the ReactiveMP.jl package, since for other compared packages it involved running the inference for more than an hour.

Table 3: A comparison of run-time duration in milliseconds for automated Bayesian inference for a hidden Markov model (13) across different methods: reactive variational message passing (ReactiveMP.jl), scheduled variational message passing (ForneyLab.jl) and a combination of Hamiltonian Monte Carlo and particle Gibbs (Turing.jl). The values in the table show the minimum possible duration across multiple runs. The T column represents the number of observation in a data set. In this benchmark the number of categories M of observations is set to 3. ReactiveMP.jl and ForneyLab.jl perform VMP on a full graph. The number of VMP iterations is set to 20. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl runs two benchmarks with 250 and 500 number of samples respectively. We present benchmark results for more than 10 000 observations only for the ReactiveMP.jl package since for other compared packages it involved running the inference for more than an hour.

T	Reactive VMP	Scheduled VMP		HMC + PG	
		Compilation	Inference	250 samples	500 samples
100	26	82280	93	93956	186293
250	80	405949	228	396946	783850
10000	7014	—	—	—	—
20000	19142	—	—	—	—

estimate their corresponding posterior distributions. For online learning in the HGF model (16a) and (16b), we define only a single time step and specify the structured factorization assumption for state transition nodes as well as for higher layer random walk transition nodes and the mean-field assumption for other functional dependencies in the model

$$q\left(s^{(1)}, s^{(2)}\right) = q\left(s^{(1)}\right)q\left(s^{(2)}\right), \tag{18a}$$

$$q\left(s^{(1)}\right) = \frac{\prod_{t=2}^{T} q\left(s_{t-1}^{(1)}, s_t^{(1)}\right)}{\prod_{t=2}^{T} q\left(s_t^{(1)}\right)}, \tag{18b}$$

$$q\left(s^{(2)}\right) = \frac{\prod_{t=2}^{T} q\left(s_{t-1}^{(2)}, s_t^{(2)}\right)}{\prod_{t=2}^{T} q\left(s_t^{(2)}\right)}. \tag{18c}$$

We show an example of an HGF model specification in Listing 16. For the approximation of the integral in the message update rule, the **GCV** node requires a suitable approximation method to be specified during model creation. For this reason, we create a **metadata** object called **GCVMetadata()** that accepts a **GaussHermiteCubature()** approximation method with a prespecified number of sigma points **gh_n**.

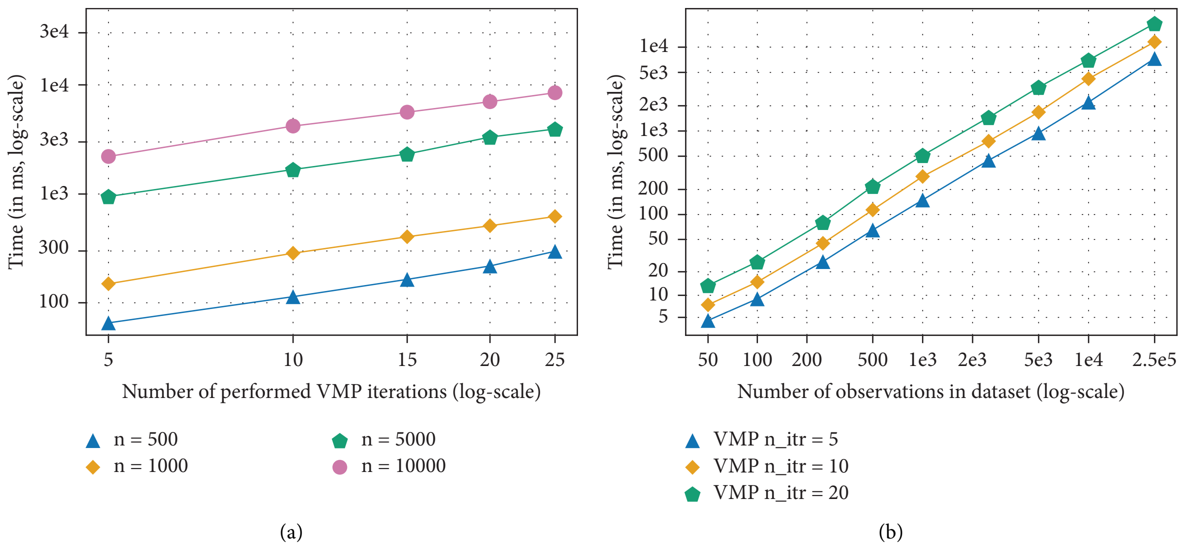


FIGURE 15: Benchmark results for the hidden Markov model in Listing 15. (a) Run-time duration vs. number of VMP iterations for different number of observations in the data set n. (b) Run-time duration vs. number of observations in the data set for a different number of VMP iterations n_itr.

TABLE 4: Comparison of posterior results accuracy in terms of metric (13) in the hidden Markov model (14) across different methods: variational message passing (ReactiveMP.jl and ForneyLab.jl) and a combination of Hamiltonian Monte Carlo and particle Gibbs (Turing.jl). Lower values indicate better performance. In this experiment, the number of categories M of observations is set to 3. ReactiveMP.jl and ForneyLab.jl perform VMP on a full graph. The number of VMP iterations is set to 20. Turing.jl runs two benchmarks with 250 and 500 numbers of samples, respectively.

	Number of observations		
	50	100	250
VMP (20 iterations)	0.10	0.09	0.07
HMC + PG (250)	0.50	0.51	0.52
HMC + PG (500)	0.51	0.49	0.51

6.3.2. Inference. For the inference procedure, we adopt the technique from the Infinite Recursive Chain Processing section (Section 4.2.5). We subscribe to future observations, perform a prespecified number of VMP iterations for each new data point, and redirect the last posterior update as prior for the next future observation. The resulting inference procedure reacts to new observations autonomously and is compatible with infinite data streams. We show a part of the inference procedure for the model (17a) and (17b) in Listing 17. The HMC algorithm has been executed similarly, by running inference for each data point and using the posterior results as corresponding priors for the next time step. Full code is available at GitHub experiment's repository.

We show the inference results in Figure 16. Figure 16(c) shows the convergence of Bethe free energy after several number of VMP iterations. The qualitative results in Figure 16(a) and in Figure 16(b) show a correct posterior estimation of continuously valued hidden states even though the model contains nonconjugate relationships among variables.

6.3.3. Benchmark. The main benchmark results are presented in Figure 17 and Table 5, and the comparison of accuracy in Table 6. We show the performance of the ReactiveMP.jl package based on the number of observations and the number of VMP iterations for this particular model in Figure 18. As shown in the previous example, we note that the ReactiveMP.jl framework scales linearly both with the number of observations and with the number of VMP iterations performed.

We can see that, in contrast with previous examples where we performed inference on a full graph, the ForneyLab.jl compilation time is no longer dependent on the number of observations, and the model compilation is more acceptable due to the fact that we always build a single time step of a graph and reuse it during online learning. Both the ReactiveMP.jl and ForneyLab.jl show the same scalability and posterior accuracy results, as they both use the same method for posterior approximation; however, ReactiveMP.jl is faster in VMP inference execution in absolute timing.

In this model, the HMC algorithm shows less accurate results in terms of the metric (12). However, due to the online learning setting, the model has a small number of unknowns, making it more feasible for the HMC algorithm to perform inferences for a large number of observations compared to the previous examples.

```
# We use "MeanField" factorization assumption
# In principle, the model specification function may accept the
# number of layers as an argument and construct the graph for any given layer
@model [ default_factorization = MeanField () ]
function hierarchical_gaussian_filter_model (gh_n, s_2_w, y_w, kappa, omega)
    # "s_2" refers to the second layer in the hierarchy
    # "s_1" refers to the first layer in the hierarchy
    s_2_prior = datavar (Float64, 2)
    s_1_prior = datavar (Float64, 2)
    y = datavar (Float64)
    s_2_previous ~ GaussianMeanPrecision (z_2_prior [1], z_2_prior [2])
    s_1_previous ~ GaussianMeanPrecision (s_1_prior [1], s_1_prior [2])
    # Z–layer modeled as a random walk with structured factorization assumption
    s_2 ~ GaussianMeanPrecision (s_2_previous, s_2_w) where {
        q = q (s_2_previous, s_2) q (s_2_w)
    }
    # GCV node uses Gauss–Hermite cubature to approximate the nonlinearity
    # between layers in the hierarchy. We may change the number of points
    # used in the approximation with the "gh_n" model argument
    meta = GCVMetadata (GaussHermiteCubature (gh_n))
    # Comma syntax for the tilde operator allows us to extract a reference to
    # the GCV node, which we will use later on to initialise joint marginals
    gcv, s_1~GCV (s_1_previous, s_2, kappa, omega) where {
        q = q (s_1, s_1_previous) q (s_2) q (kappa) q (omega), meta = meta
    }
    y ~ GaussianMeanPrecision (s_1, y_w)
    return s_2_prior, s_1_prior, gcv, s_2, s_1, y
end
```

Listing 16: An example of model specification for the hierarchical Gaussian filter model (17).

```
# This function will be called every time we observe a new data point
function on_next! (actorHGFInferenceActor, dataFloat64)
    s_2_prior, s_1_prior, gcv, s_2, s_1, y = actor.model_output
    # To perform multiple VMP iterations we pass the data multiple times
    # It forces the inference backend to react to the data and to
    # update posterior marginals multiple times
    for i in 1:actor.n_vmp_iterations
            update! (s_2_prior, actor.current_s_2)
            update! (s_1_prior, actor.current_s_1)
            update! (y, data)
    end
    . . .
    # Update current posterior marginals at time step "t"
    actor.current_s_2 = mean_precision (last (actor.history_s_2))
    actor.current_s_1 = mean_precision (last (actor.history_s_1))
end
```

Listing 17: A part of an example of inference specification for the hierarchical Gaussian filter model (16a) and (16b).

7. Discussion

We are investigating several possible future directions for the reactive message passing framework that could improve its usability and performance. We believe that the current implementation is efficient and flexible enough to perform real-time reactive Bayesian inference and has been tested on both synthetic and real data sets. The new framework allows large models to be applied in practice and simplifies the model exploration process in signal processing applications. The proposed architecture does not create any explicit schedule and simply allows nodes and edges to react on local changes in their Markov blanket. The lack of explicit scheduling comes with a number of useful properties. The inference engine does not need to traverse the entire factor graph, which may take a significant amount of computer resources. There is no need to rebuild the schedule in case of a model structure change or a data sensor failure. As a result,

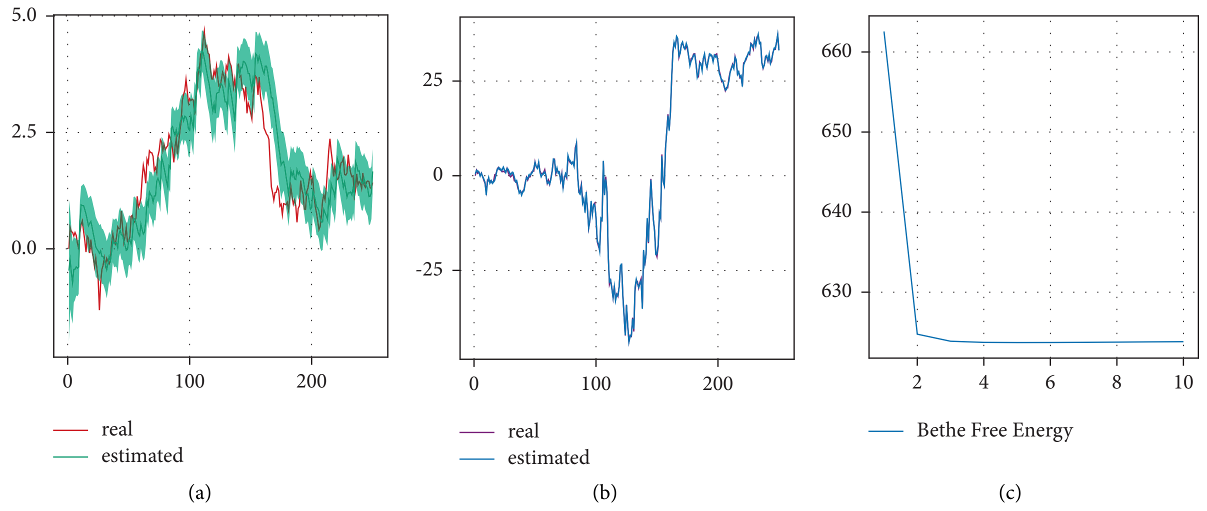


FIGURE 16: Online learning inference results for the Hierarchical Gaussian filter model (16a) and (16b) for 250 synthetically generated 1-dimensional observations. (a) The inference results of layer $s_t^{(2)}$ hidden states. The x-axis represents the time steps and the y-axis corresponds to the actual value of hidden states. (b) The inference results of layer $s_t^{(1)}$ hidden states. The x-axis represents the time steps and the y-axis corresponds to the actual value of hidden states. (c) Bethe free energy evaluation results. The x-axis represents an index of VMP iteration. The y-axis represents a Bethe Free Energy value averaged over all data points at a specific VMP iteration.

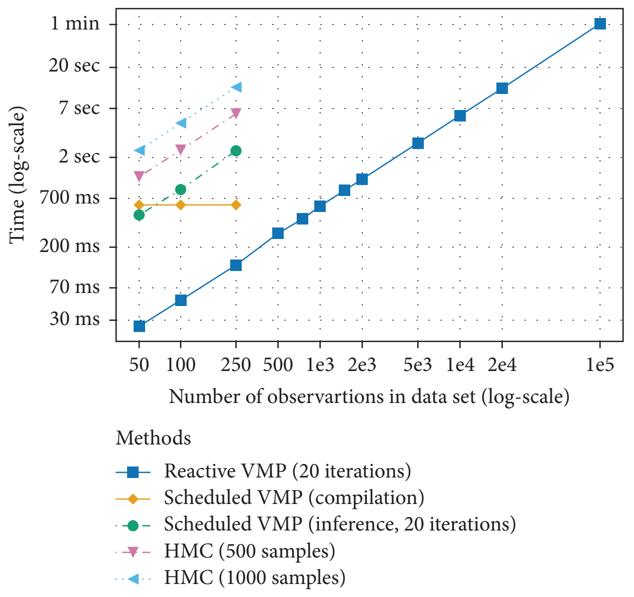


FIGURE 17: A comparison of run-time duration in milliseconds for automated Bayesian inference in a Hierarchical Gaussian filter model (16a) and (16b) across different methods: reactive variational message passing (ReactiveMP.jl), scheduled variational message passing (ForneyLab.jl) and Hamiltonian Monte Carlo (Turing.jl). The values in the figure show the minimum possible duration across multiple runs. ReactiveMP.jl and ForneyLab.jl perform online learning with VMP on a single time step of the corresponding graph. The number of VMP iterations performed is set to 20. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl uses HMC sampling with 500 and 1000 number of samples respectively.

inference execution is adaptive and more robust. Without a fixed message passing scheduling, it is possible to update messages at different rates in different parts of the model graph. All of these properties arise naturally in the proposed architecture and come for "free." The actual order, in which messages will be executed, is data-dependent and results from which nodes or edges had an opportunity to react first. Crucially, there is no need to specify the order explicitly. As mentioned in Section 6, the resulting architecture has been successfully battle-tested in sophisticated probabilistic models for real-world scenarios.

A natural future direction is to apply the new framework to an *Active Inference* [45] setting. A reactive active inference agent learns purposeful behavior solely through situated interactions with its environment and processing these interactions by real-time inference. As we discussed in Motivation (Section 2), an important issue for autonomous agents is robustness of the running Bayesian inference processes, even when nodes or edges collapse under situated conditions. ReactiveMP.jl supports in-place model adjustments during the inference procedure as well as handles missing data observations, but it does not export any user-friendly API yet. For the next release of our framework, we aim to export a public API for robust Bayesian inference to simplify the development of active inference agents. In addition, our plan is to proceed with further optimization of the current implementation and improve the scalability of the existing package in real time on embedded devices. Moreover, the Julia programming language is developing and improving, and thus, we expect it to be even more efficient in the coming years.

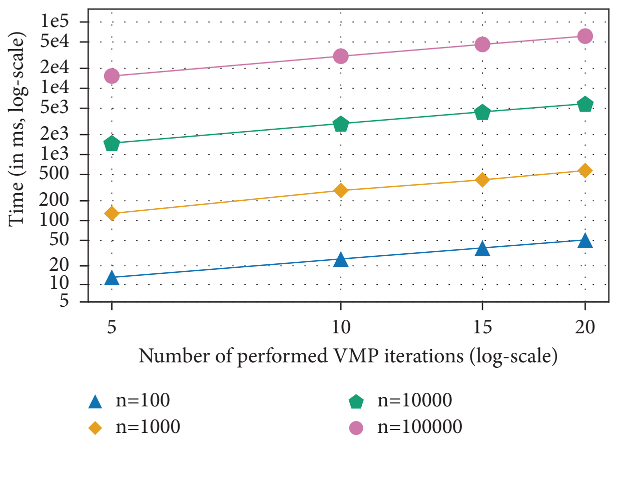
Reactive programming allows us to easily integrate additional features into our new framework. First, we

Table 5: A comparison of run-time duration in milliseconds for automated Bayesian inference in a Hierarchical Gaussian Filter model (16a) and (16b) across different methods: reactive variational message passing (ReactiveMP.jl), scheduled variational message passing (ForneyLab.jl) and Hamiltonian Monte Carlo (Turing.jl). The values in the figure show the minimum possible duration across multiple runs. The T column represents number of observation in a data set. ReactiveMP.jl and ForneyLab.jl perform online learning with VMP on a single time step of the corresponding graph. The number of VMP iterations performed is set to 20. The ReactiveMP.jl timings include graph creation time. The ForneyLab.jl pipeline consists of model compilation, followed by actual inference execution. Turing.jl uses HMC sampling with 500 and 1000 number of samples respectively.

T	Reactive VMP	Scheduled VMP		HMC	
		Compilation	Inference	500 samples	1000 samples
50	19	597	457	1357	3036
100	38	602	876	2607	5287
250	95	592	2368	6531	12847
10000	4412	—	—	—	—
100000	45889	—	—	—	—

Table 6: Comparison of posterior results accuracy in terms of metric (12) in the hierarchical Gaussian filter model (17b) across different methods: variational message passing (ReactiveMP.jl and ForneyLab.jl) and Hamiltonian Monte Carlo (Turing.jl). Lower values indicate better performance. ReactiveMP.jl and ForneyLab.jl perform online learning with VMP on a single time step of the corresponding graph. The number of VMP iterations is set to 20. Turing.jl runs two benchmarks with 500 and 1000 number of samples, respectively.

	Number of observations					
	$x_k^{(2)}$ layer			$x_k^{(1)}$ layer		
	50	100	250	50	100	250
VMP (20 iterations)	0.89	0.79	0.75	0.36	0.35	0.35
HMC (500)	1.41	1.29	1.51	0.45	0.62	4.74
HMC (1000)	1.30	1.14	1.22	0.41	0.49	2.56



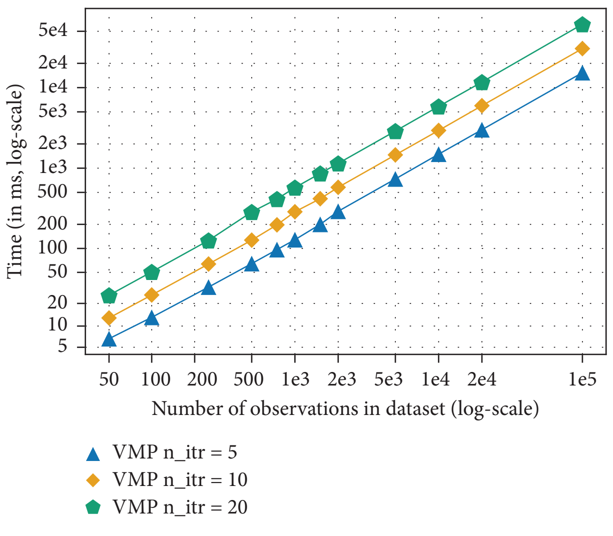


Figure 18: Benchmark results for the Hierarchical Gaussian filter model (16a) and (16b) in Listing 16. (a) Benchmark: run-time duration vs. number of VMP iterations for different number of observations in data set n. (b) Benchmark: run-time duration vs number of observations in the data set for different number of VMP iterations n_itr.

investigate the possibility of running reactive message passing-based inference in parallel on multiple CPU cores [46]. RP does not make any assumptions in the underlying data generation process and does not distinguish between synchronous and asynchronous data streams. Second, reactive systems allow us to integrate local stopping criteria for passing messages. In some scenarios, we may want to stop passing messages based on some prespecified criterion and simply stop reacting to new observations and save battery life on a portable device or to save computational power for other tasks. Third, our current implementation does not yet provide extensive debugging tools, but it might be crucial to analyze the performance of message passing-based methods step-by-step. We are looking at options to extend the graphical notation and integrate useful debug methods with the new framework to analyze and debug message passing-based inference algorithms visually in real time and explore their performance characteristics. Since the ReactiveMP.jl package uses reactive observables under the hood, it should be straightforward to "spy" on all updates in the model for later or even real-time performance analysis. That should even further simplify the process of model specification, as one may change the model in real time and immediately see the results.

Moreover, we are in a preliminary stage of extending the set of available message passing rules to a broader class of nonconjugate priors and likelihood pairs [30]. Generic probabilistic toolboxes in the Julia language, like Turing.jl, support inference in a broader range of probabilistic models, which is currently not the case for the ReactiveMP.jl. We are working on the future extension of message update rules and potential integration with other probabilistic frameworks in the Julia community.

Finally, another interesting future research direction is to decouple the model specification language from factorization and the form constraint specification in the variational family of distributions $\mathcal{Q}_B$. This would allow us to have a single model $p(s, y)$ and a set of constrained variational families of distributions $\mathcal{Q}_{B_i}$ so we could run and compare different optimization procedures and their performance simultaneously or trade off computational complexity with accuracy in real time.

8. Conclusions

In this paper, we presented methods and implementation aspects of a reactive message passing (RMP) framework both for exact and approximate Bayesian inference, based on minimization of a constrained Bethe free energy functional. The framework is capable of running hybrid algorithms with BP, EP, EM, and VMP analytical message update rules and supports local factorization as well as form constraints on the variational family. We implemented an efficient proof-of-concept of RMP in the form of the ReactiveMP.jl package for the Julia programming language, which exports an API for running reactive Bayesian inference and scales easily to large state-space models with hundreds of thousands of unknowns. The inference engine is highly customizable through ad hoc construction of custom nodes, message update rules, approximation methods, and optional modifications to the default computational pipeline. The experimental results for a set of standard signal processing models indicated better performance for RxInfer relative to other Julia packages for automated Bayesian inference. The benchmark results showed that the overhead, associated with managing the reactive nature of the proposed architecture, is minimal and that RMP generally outperforms the reference message passing-based implementation. We believe that the reactive programming approach to message passing-based methods opens a lot of directions for further research and will bring real-time Bayesian inference closer to real-world applications.

Data Availability

The datasets generated during and/or analysed during the current study are available in the biaslab/ReactiveMPPaperExperiments repository, https://github.com/biaslab/ReactiveMPPaperExperiments.

Disclosure

The pre-print of the current work is available at arXiv Bagaev and de Vries [47].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Authors' Contributions

All authors contributed to the study conception and design. Dmitry Bagaev performed material preparation, data collection, experimental evaluation, and analysis of the results. Dmitry Bagaev wrote the first draft of the manuscript, and all authors commented on previous versions of the manuscript. All authors read and approved the final manuscript.

Acknowledgments

The authors would like to acknowledge the assistance and their contributions to the source code of the ReactiveMP.jl package from Ismail Senoz, Albert Podusenko, and Bart van Erp and the support and helpful discussions from Thijs van de Laar, one of the creators of the ForneyLab.jl package. The authors are also grateful to all BIASlab team members for their support during the early stages of the development of this project. This work was partially supported by a TKI grant from the Netherlands Enterprise Agency (RVO) and by a grant from hearing aid manufacturer GN Hearing.

References

[1] M. Abadi, A. Agarwal, B. Paul et al., "TensorFlow: large-scale machine learning on heterogeneous systems," 2015, https://arxiv.org/abs/1603.04467.

[2] P. Adam, S. Gross, F. Massa et al., "PyTorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, vol. 32, pp. 8024–8035, Curran Associates, Inc, Red Hook, NY, USA, 2019.

[3] K. Friston, "The free-energy principle: a rough guide to the brain?" *Trends in Cognitive Sciences*, vol. 13, no. 7, pp. 293–301, 2009.

[4] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[5] E. Gal, I. McGraw, and D. Koller, "Residual belief propagation: informed scheduling for asynchronous message passing," 2012, https://arxiv.org/ftp/arxiv/papers/1206/1206.6837.pdf.

[6] P. Radosavljevic, A. de Baynast, and J. R. Cavallaro, "Optimized message passing schedules for LDPC decoding," in *Proceedings of the Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers,*

2005, pp. 591–595, IEEE, Pacific Grove, CA, USA, October 2005.

[7] E. Sharon, S. Litsyn, and J. Goldberger, "An efficient message-passing schedule for LDPC decoding," in *Proceedings of the 2004 23rd IEEE Convention of Electrical and Electronics Engineers in Israel*, pp. 223–226, IEEE, Tel-Aviv, Israel, September 2004.

[8] C. Knoll, M. Rath, S. Tschiatschek, and F. Pernkopf, "Message scheduling methods for belief propagation," in *Annalisa Appice, Pedro Pereira Rodrigues, Vítor Machine Learning and Knowledge Discovery in Databases, volume 9285*, S. Costa, J. Gama, A. Jorge, and C. Soares, Eds., pp. 295–310, Springer International Publishing, New York, NY, USA, 2015.

[9] E. Bainomugisha, A. L. Carreton, T. V. Cutsem, S. Mostinckx, and W. D. Meuter, "A survey on reactive programming," *ACM Computing Surveys*, vol. 45, no. 4, pp. 1–34, 2013.

[10] F. Boussinot, B. Monasse, and J.-F. Susini, "Reactive programming of simulations in physics," *International Journal of Modern Physics C*, vol. 26, no. 12, Article ID 1550132, 2015.

[11] J. Busch, J. Pi, and T. Seidl, "PushNet: Efficient and adaptive neural message passing," in *Proceedings of the 24th European Conference on Artificial Intelligence, volume 325 of Frontiers in Artificial Intelligence and Applications*, IOS Press, Amsterdam, Netherlands, December 2020.

[12] G. Baudart, L. Mandel, E. Atkinson, B. Sherman, M. Pouzet, and M. Carbin, "Reactive probabilistic programming," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2020*, pp. 898–912, Association for Computing Machinery, New York, NY, USA, April 2020.

[13] P. Albert, Bart van Erp, D. Bagaev, Ş. İsmail, and Bert de Vries, "Message passing-based inference in the Gamma mixture model," in *Proceedings of the IEEE International Workshop on Machine Learning for Signal Processing*, pp. 1–6, IEEE, Gold Coast, Australia, October 2021.

[14] B. van Erp, A. Podusenko, T. Ignatenko, and B. de Vries, "A Bayesian modeling approach to situated design of personalized soundscaping algorithms," *Applied Sciences*, vol. 11, no. 20, p. 9535, 2021.

[15] A. Podusenko, B. van Erp, M. Koudahl, and B. de Vries, "AIDA: an active inference-based design agent for audio processing algorithms," *Frontiers in Signal Processing*, vol. 2, 2022.

[16] W. Kouw, P. Albert, M. Koudahl, and M. Schoukens, "Variational message passing for online polynomial NARMAX identification," April 2022, https://arxiv.org/abs/2204.00769.

[17] G. Forney, "Codes on graphs: normal realizations," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 520–548, 2001.

[18] H.-A. Loeliger, "An introduction to factor graphs," *IEEE Signal Processing Magazine*, vol. 21, no. 1, pp. 28–41, 2004.

[19] S. Korl, "A factor graph approach to signal modelling, system identification and filtering," PhD Thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 2005.

[20] M. B. Christopher, *Pattern Recognition And Machine Learning*, Springer-Verlag New York, Inc, New York, NY, USA, 2006.

[21] J. Winn, "Variational message passing and its applications," PhD Thesis, University of Cambridge, Cambridge, England, 2003.

[22] J. Dauwels, "On variational message passing on factor graphs," in *Proceedings of the IEEE International Symposium on Information Theory*, pp. 2546–2550, Nice, France, June 2007.

[23] C. Zhang, J. Butepage, H. Kjellstrom, and S. Mandt, "Advances in variational inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, no. 8, pp. 2008–2026, 2019.

[24] I. Şenöz, T. van de Laar, B. de Vries, and D. Bagaev, "Variational message passing and local constraint manipulation in factor graphs," *Entropy*, vol. 23, no. 7, p. 807, 2021.

[25] J. Yedidia, W. Freeman, and Y. Weiss, "Bethe free energy, Kikuchi approximations, and belief propagation algorithms," *Advances in Neural Information Processing Systems*, vol. 13, no. 24, 2001.

[26] J. S. Yedidia, W. T. Freeman, and Y. Weiss, "Constructing free-energy approximations and generalized belief propagation algorithms," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2282–2312, 2005.

[27] D. Zhang, X. Song, W. Wang, G. Fettweis, and X. Gao, "Unifying message passing algorithms under the framework of constrained Bethe free energy minimization," *IEEE Transactions on Wireless Communications*, vol. 20, no. 7, pp. 4144–4158, 2021.

[28] R. Kuhn, B. Hanafee, and J. Allen, *Reactive Design Patterns*, Manning Publications Co, Shelter Island, NY, USA, 1 edition, 2017.

[29] S. Akbayrak and B. de Vries, "Reparameterization gradient message passing," in *Proceedings of the 27th European Signal Processing Conference (EUSIPCO)*, Coruna, Spain, September 2019.

[30] S. Akbayrak, I. Bocharov, and B. de Vries, "Extended variational message passing for automated approximate Bayesian inference," *Entropy*, vol. 23, no. 7, p. 815, 2021.

[31] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: a fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[32] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," 2012, https://arxiv.org/abs/1209.5145.

[33] Thijs van de Laar, M. Cox, and B. de Vries, *ForneyLab.jl: A Julia Toolbox for Factor Graph-Based Probabilistic Programming*, JuliaCon, London, UK, 2018.

[34] H. Ge, K. Xu, and G. Zoubin, "Turing: A language for flexible probabilistic inference," in *International Conference on Artificial Intelligence and Statistics*, pp. 1682–1690, PMLR, New York City, NY, USA, 2018.

[35] J. Chen, J. Revels, and A. Edelman, "Robust benchmarking in noisy environments," in *HPEC'16 Proceedings of the 20th IEEE High Performance Extreme Computing Conference*, IEEE, Waltham, MA, USA, 2016.

[36] S. Simo, *Bayesian Filtering and Smoothing*, Cambridge University Press, Cambridge, England, 2013.

[37] S. SärkkÄ, "Unscented rauch–tung–striebel smoother," *IEEE Transactions on Automatic Control*, vol. 53, no. 3, pp. 845–849, 2008.

[38] J. Frederick, *Statistical Methods for Speech Recognition*, MIT Press, Cambridge, MA, USA, 1998.

[39] L. Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition,* PTR Prentice Hall, Hoboken, NJ, USA, 1993.

[40] M. Christopher and S. Hinrich, *Foundations Of Statistical Natural Language Processing*, The MIT Press, Cambridge, MA, USA, 1999.

[41] Z. Zhang, D. Wang, G. Dai, and M. I. Jordan, "Matrix-variate Dirichlet process priors with applications," *Bayesian Analysis*, vol. 9, no. 2, pp. 259–286, 2014.

[42] C. D. Mathys, E. I. Lomakina, J. Daunizeau et al., "Uncertainty in perception and the hierarchical Gaussian filter," *Frontiers in Human Neuroscience*, vol. 8, p. 825, 2014.

[43] I. Şenoz and B. De Vries, “Online state and parameter estimation in the hieararchical gaussian filter,” in *Proceedings of the IEEE International Workshop on Machine Learning for Signal Processing*, Los Angeles, CA, USA, June 2020.

[44] J. Kokkala, A. Solin, and S. Simo, “Sigma-point filtering and smoothing based parameter estimation in nonlinear dynamic systems,” *Journal of Advances in Information Fusion*, vol. 11, no. 1, pp. 15–30, 2016.

[45] K. Friston, T. FitzGerald, F. Rigoli, P. Schwartenbeck, J. O’Doherty, and G. Pezzulo, “Active Inference and Learning,” *Neuroscience & Biobehavioral Reviews*, vol. 68, 2016.

[46] S. Sarkka and A. F. Garcia-Fernandez, “Temporal parallelization of bayesian smoothers,” *IEEE Transactions on Automatic Control*, vol. 66, no. 1, pp. 299–306, 2021.

[47] D. Bagaev and B. De Vries, “Reactive message passing for scalable bayesian inference,” 2021, https://arxiv.org/abs/2112.13251.