Turing.jl: a general-purpose probabilistic programming language

Turing.jl: a general-purpose probabilistic programming language

TOR ERLEND FJELDE, University of Cambridge, United Kingdom
KAI XU, MIT-IBM Watson AI Lab, USA
DAVID WIDMANN, Pumas-AI Inc., USA
MOHAMED TAREK, Pumas-AI Inc., USA and Business School, University of Sydney, Australia
CAMERON PFIFFER, Stanford University, USA
MARTIN TRAPP, Aalto University, Finland
SETH D. AXEN, University of Tübingen, Germany
XIANDA SUN, University of Cambridge, United Kingdom
MARKUS HAURU, PENELOPE YONG, and WILL TEBBUTT, The Alan Turing Institute, United Kingdom
ZOUBIN GHAHRAMANI, Google DeepMind and University of Cambridge, United Kingdom
HONG GE*, University of Cambridge, United Kingdom

Probabilistic programming languages (PPLs) are becoming increasingly important in many scientific disciplines, such as economics, epidemiology, and biology, to extract meaning from sources of data while accounting for one's uncertainty. The key idea of probabilistic programming is to decouple inference and model specification, thus allowing the practitioner to approach their task at hand using Bayesian inference, without requiring extensive knowledge in programming or computational statistics. At the same time, the complexity of problem settings in which PPLs are employed is steadily increasing, both in terms of project size and model complexity, calling for more flexible and efficient systems.

In this work, we describe `Turing.jl`, a general-purpose PPL, which is designed to be flexible, efficient, and easy to use. `Turing.jl` is built on top of the Julia programming language, which is known for its high performance and ease-of-use. We describe the design of `Turing.jl`, contextualizing it within different types of users and use cases, its key features, and how it can be used to solve a wide range of problems. We also provide a brief overview of the ecosystem around `Turing.jl`, including the different libraries and tools that can be used in conjunction with it. Finally, we provide a few examples of how `Turing.jl` can be used in practice.

*To whom correspondence should be addressed.

Authors' addresses: Tor Erlend Fjelde, tef30@cam.ac.uk, University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, CB3 0FD; Kai Xu, MIT-IBM Watson AI Lab, USA, xuk@mit.edu; David Widmann, Pumas-AI Inc., USA, d.widmann@pumas.ai; Mohamed Tarek, Pumas-AI Inc., USA and Business School, University of Sydney, Australia, mohamed@pumas.ai; Cameron Pfiffer, Stanford University, USA, cpfiffer@stanford.edu; Martin Trapp, Aalto University, Department of Computer Science, Espoo, Finland, martin.trapp@aalto.fi; Seth D. Axen, University of Tübingen, Tübingen, Germany, seth@sethaxen.com; Xianda Sun, xs307@cam.ac.uk, University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, CB3 0FD; Markus Hauru, markus@mhauru.org; Penelope Yong, pyong@turing.ac.uk; Will Tebbutt, wtebbutt@turing.ac.uk, The Alan Turing Institute, United Kingdom; Zoubin Ghahramani, zoubin@eng.cam.ac.uk, Google DeepMind and University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, CB3 0FD; Hong Ge, hg334@cam.ac.uk, University of Cambridge, Cambridge, Cambridgeshire, United Kingdom, CB3 0FD.

ACM 2836-8924/2025/2-ART
https://doi.org/10.1145/3711897

CCS Concepts: • **Computing methodologies** → **Uncertainty quantification**; **Modeling methodologies**; *Learning in probabilistic graphical models*; **Latent variable models**; *Maximum a posteriori modeling*; • **Software and its engineering** → **Software libraries and repositories**; • **Mathematics of computing** → **Bayesian computation**; **Variational methods**; **Markov-chain Monte Carlo methods**; *Sequential Monte Carlo methods*.

Additional Key Words and Phrases: Probabilistic Programming, Probabilistic Programming Languages, Probabilistic Inference, Bayesian Inference, Markov Chain Monte Carlo, Variational Inference, Sequential Monte Carlo, Uncertainty Quantification, Modeling Methodologies, Latent Variable Models, Maximum a Posteriori Modeling, Software Libraries and Repositories, Bayesian Computation, Variational Methods, Sequential Monte Carlo Methods.

1 INTRODUCTION

Probabilistic programming languages (PPLs) extend common programming paradigms with probabilistic semantics that allows one to specify stochastic programs. PPLs typically form a broad class of software tools that allow users to specify probabilistic models as computer programs (i.e. codes) in a high-level programming language accessible to domain experts. The models specified by PPLs can be used, in an automated manner, to perform probabilistic inference, i.e., to compute the posterior distribution of latent (unobserved) variables in the model given observed data. This posterior distribution can then be used for a variety of tasks, such as making predictions about future data or learning about specific parameters of interest.

In this work, we describe the PPL `Turing.jl`, a PPL embedded in the Julia programming language [8]. `Turing.jl` was first introduced in [31] and has since then seen significant adoption and continued development by the community [4, 11, 27, 49, 71, 73, 81, 86]. `Turing.jl` is a somewhat unique PPL as it is embedded in Julia and benefits from its capabilities, avoiding the use of transpilation to a lower-level language or custom compilers to achieve high performance. Instead, `Turing.jl` can utilize the Julia compiler to generate efficient code for the probabilistic models. This has the advantage of making it easier to develop and maintain the PPL, as it can leverage the existing ecosystem of Julia packages for scientific computing and machine learning, all the while providing competitive performance with other PPLs. We will first outline the basics of probabilistic inference and programming in Section 1, contextualizing the needs of a PPL by categorizing the different types of users of PPLs in Section 1.3. Subsequently, `Turing.jl` will be introduced and described at a high level in Section 2 and how it fits into a standard Bayesian workflow (Figure 1), including demonstrations of some interesting examples of probabilistic models that can be expressed in `Turing.jl` in Section 3. Then in Section 4, the internals of `Turing.jl` are described, including design choices made and how they impact performance and usability for the different types of users, before we finally discuss `Turing.jl` in the context of other PPLs.

1.1 Probabilistic modeling and inference

In both scientific and industrial applications, the typical (Bayesian) data analysis pipeline involves fitting a series of models on a collection of data $\mathcal{D} = (y_i)_{i=1}^N$ to perform inference about, for example, the model (latent) parameters, future events, or even the models themselves. It is often desirable to additionally incorporate expert knowledge into this process, which may be expressed in the form of a distribution over the model parameters. This is the domain of *probabilistic modeling and inference*.

In probabilistic modeling we usually start by writing down how we go from the parameters θ to the data $\mathcal{D}$, which is encapsulated in the idea of a *likelihood* function $p(\mathcal{D} \mid \theta)$, quantifying how probable the observed data $\mathcal{D}$ is given the parameters θ. The next step is to incorporate our prior knowledge about the parameters θ into the *prior* distribution $p(\theta)$, describing our beliefs about the parameters before seeing the data. The prior distribution is then combined with the likelihood function to form the *posterior* distribution through Bayes' theorem:

$$p(\theta \mid \mathcal{D}) = \frac{p(\mathcal{D}, \theta)}{p(\mathcal{D})} = \frac{p(\mathcal{D} \mid \theta)p(\theta)}{p(\mathcal{D})}, \tag{1}$$

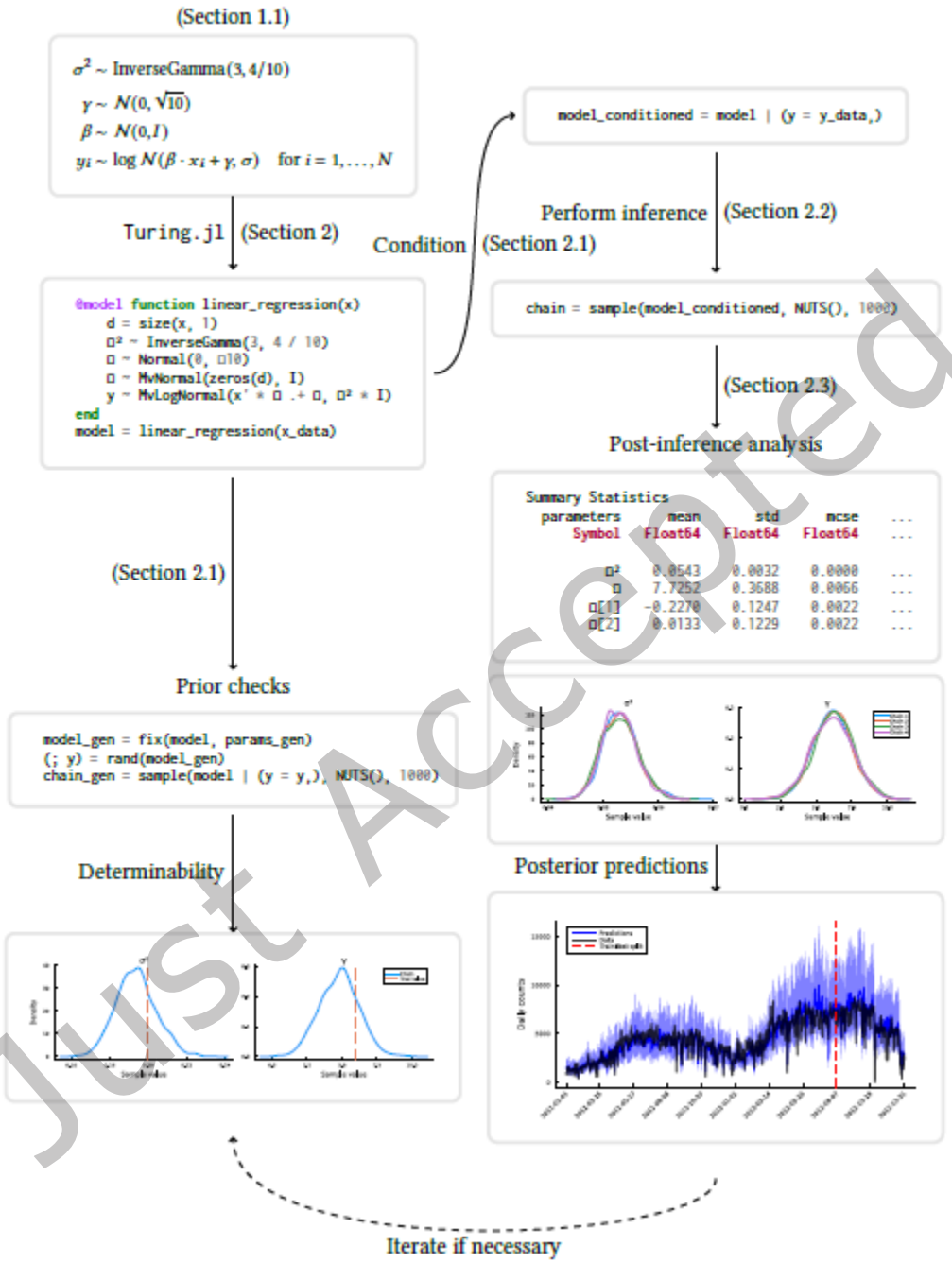

Fig. 1. A brief illustration of a typical Bayesian workflow with `Turing.jl`. See referenced sections for details.

describing our beliefs about the parameters *after* observing data. *Bayesian inference* refers to the process of characterizing this posterior given prior, likelihood, and data.

In (1) the posterior distribution is expressed through the conditional distribution $p(\theta \mid \mathcal{D})$, but when writing down a probabilistic model we usually employ the *generative process* notation:

$$\theta \sim p(\theta)$$
$$\mathcal{D} \sim p(\mathcal{D} \mid \theta)$$

Here $\sim$ represents the "is distributed as" relation, and so the above expresses the *joint* distribution $p(\mathcal{D}, \theta)$ without any conditioning on the data $\mathcal{D}$; both θ and $\mathcal{D}$ are treated as random variables. Only when we want to perform inference, i.e., inspect the posterior $p(\theta \mid \mathcal{D})$, we specify that the data $\mathcal{D}$ in the generative process is "observed" and to be conditioned on to produce the posterior distribution $p(\theta \mid \mathcal{D})$.

To make things more concrete, let's consider a simple example model and the corresponding inference tasks we might want to perform.

Example 1.1 (Bayesian linear regression). In this example, we are given a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^N$ of bike sharing data [28], consisting of daily counts of bike rentals $(y_i)_{i=1}^N$ along with weather and seasonal information as input features $(x_i)_{i=1}^N$. The goal is to predict the number of bike rentals y based on the weather and seasonal information x, while quantifying the uncertainty in our estimates. One approach to modeling this setting is to use a linear regression model, where the *log* of the number of bike rentals is modeled as a linear function of the weather and seasonal information. The resulting model can be described through the following generative process for the joint $p(\mathcal{D}, \sigma^2, \beta, \gamma)$:

$$\begin{aligned} \sigma^2 &\sim \text{InverseGamma}(3, 4/10) \\ \gamma &\sim \mathcal{N}(0, \sqrt{10}) \\ \beta &\sim \mathcal{N}(0, I) \\ y_i &\sim \log \mathcal{N}(\beta \cdot x_i + \gamma, \sigma) \quad \text{for } i = 1, \ldots, N \end{aligned} \qquad \text{(Linear regression)}$$

where σ is the standard deviation of the noise in the data, γ is the intercept of our linear model, β is the vector of coefficients for the input features $(x_i)_{i=1}^N$, $\mathcal{N}(\mu, \sigma)$ denotes a univariate Gaussian with mean μ and standard deviation σ, $\mathcal{N}(0, I)$ denotes a multivariate Gaussian with mean 0 and covariance I (identity matrix), and $\log \mathcal{N}(\mu, \sigma)$ denotes a log-Gaussian with similar conventions[1]. Here we have assumed that the noise in the data is Gaussian and homoscedastic, i.e., the variance of the noise is constant across all data points, in addition to using fairly uninformative priors for the parameters.

We will use Example 1.1 throughout this paper to illustrate different functionality and features of the probabilistic programming language `Turing.jl`. In particular, as we're interested in predicting the number of bike rentals, we will split the data into a training set $\mathcal{D}_{\text{train}}$ of length 584 and a test set $\mathcal{D}_{\text{test}}$ of length 147. Subsequently, we will use the data from the training set to obtain the posterior distribution over our model parameters and make predictions on the test set, i.e., on unseen data, using the posterior predictive distribution.

More intricate models and examples will be considered in Section 3.

1.2 Probabilistic programming languages

From the previous section we can see that performing probabilistic inference is a desirable approach for many problems, but it can be quite challenging to do so in practice for a multitude of reasons.

[1]We use the convention that in the univariate case the second argument to $\mathcal{N}$ is the *standard deviation* while in the multivariate case the second argument to $\mathcal{N}$ is the *covariance matrix*. This is to stay consistent with the notiation typically used in `Distributions.jl`, which is used extensively in `Turing.jl`, as we will see in later sections.

- *Model specification:* Writing down a probabilistic model that captures the structure of the data and the assumptions we want to make about it can be a difficult task, usually requiring a deep understanding of both the problem domain and the statistical methods. Moreover, in practice one usually must iterate on the model specification as one learns more about the problem and the data. This can be a time-consuming process if the model is not easily modifiable. Having a model specification format that is somewhat uniform across problems and easy to communicate is beneficial in this setting.
- *Model implementation:* Even if one has a well-specified model, implementing it in a programming language can be a non-trivial task, especially if one does not already have experience with the language or the tools available. This can be a significant barrier to entry for many users, especially those who are not already experienced programmers.
- *Inference implementation:* Many models of interest are complex and may involve a large number of parameters, hidden states, and/or data points. This can make it difficult or, often, impossible to derive the posterior distribution in closed form. Hence one must resort to approximate methods to perform inference, such as Markov Chain Monte Carlo (MCMC) and Variational Inference (VI). Implementing these methods requires not only an understanding of the underlying theory but also the ability to write efficient and numerically stable code.

This is where a framework with a *probabilistic programming language (PPL)* [48] can be of great help. A PPL is a high-level "programming language" that allows users to specify probabilistic models in a more intuitive and modular way, in addition to providing tools for performing inference on these models. All in all, a PPL attempts to bridge the gap between the problem domain and the computational tools, allowing the users to spend more time focusing on the modeling problem at hand rather than the implementation details.

1.3 User groups

As mentioned, PPLs are designed to be used by a wide range of users, from domain experts with little to no background in statistics or computer science to researchers who are developing new inference algorithms. In this work we find it useful to separate users of PPLs into three groups based on their interaction with a PPL:

User Group 1 (Basic user)**:** Someone who uses the PPL to solve a problem of interest to them. They may not have a background in statistics or computer science and may not be interested in the implementation details of the PPL. They mainly want to use it to solve their problem at hand. For example, an epidemiologist who wants to use a PPL to model the spread of a disease.

User Group 2 (Advanced user)**:** Someone who uses the PPL to solve a problem of interest to them but is also accustomed to implementing their own specialized inference algorithms if needed. They may be less interested in the convenience of a PPL but are interested in the flexibility it provides. For example, an applied statistician supporting a basic user (UG 1).

User Group 3 (Inference researcher)**:** Someone who works on developing new inference algorithms with potential applicability in probabilistic inference. They may not be interested in the details of the PPL but want their inference algorithms to be useful to end-users of the PPL. For example, a researcher working on a more efficient MCMC algorithm that is useful for both basic (UG 1) and advanced users (UG 2).

This categorization is not exhaustive nor are the categories mutually exclusive; one might find oneself taking several of these roles in a larger projects. It nonetheless makes it clear that there is a wide range of users of PPLs, with different needs and interests, and with mutually beneficial relationships between them. Users from (UG 1) are often from non-statistical backgrounds, e.g. biologists, who then collaborate with users from (UG 2), e.g. applied statisticians, to solve their problems. Similarly, users from (UG 1) and (UG 2) can benefit from the work of inference developers (UG 3), who in turn benefit from the feedback and use-cases provided by the

users. It is therefore important that a PPL caters to the needs of all these users: it should be easy enough to use for (UG 1), flexible enough for (UG 2), and extensible enough for (UG 3). More specifically, users from (UG 1) usually require a high-level interface to specify their models, ideally a domain-specific language (DSL) with an easy-to-understand and easy-to-use syntax. In addition, they need efficient inference algorithms[2] that work out-of-the-box, without requiring extensive tuning or knowledge of the underlying algorithms. Users from (UG 2), on the other hand, require a more flexible interface, allowing them to specify custom distributions, make use of user-defined functions, and have more control over the inference process for when the out-of-the-box inference methods are insufficient. Finally, users from (UG 3) require a flexible and extensible inference interface, allowing them to easily implement and test new inference algorithms, preferably in a way that is also useful to the other user groups.

Addressing all of these needs is a non-trivial task to do in a single PPL, hence there have been a number of different PPLs developed over the years since the pioneering work of the likes of `BUGS` [84], each with different design choices and trade-offs. In this work, we will discuss how `Turing.jl` attempts to address these needs, and how it compares to other PPLs in this regard.

Finally, there is a fourth group of "users", the developers of the PPL itself.

User Group 4 (PPL developer)**:** Developers of the PPL itself, who are responsible for implementing the PPL and maintaining it. This involves everything from implementing the core inference algorithms to designing the DSL and the high-level interface, in addition to maintaining the code base and ensuring that it is easy to extend and modify. They need to balance the needs of the other user groups and ensure that the PPL is useful to all of them.

This is less of a user group in the traditional sense, but in the end they are the ones who need to develop and maintain the PPL, which also needs to be considered in the design of a PPL. Although this group is not the main target audience of this manuscript, some sections are written with this user group in mind, in hopes that it can provide some useful insights and ideas for the development of PPLs.

With the user groups in mind, we point users to the corresponding sections in the text, depending on what we believe to be most relevant.

- (UG 1) Recommended: Section 2.1, Section 2.2, Section 2.3, and Section 3. Optional: Section 2.4, Section 2.5 and Section 5.
- (UG 2) Recommended: Section 2, Section 3 and Section 5. Optional: Section 4.
- (UG 3) Recommended: Section 2.1, Section 2.2, Section 4.3. Optional: Section 2, and Section 5.
- (UG 4) Recommended: Section 2.1, Section 4, and Section 5. Optional: Section 2.

2 TURING.JL

`Turing.jl` is a PPL implemented in the Julia programming language. It was first introduced in the work of Ge et al. [31] and has since then seen both significant adoption and continued development [4, 11, 27, 49, 71, 73, 81, 86]. `Turing.jl` is designed to be a flexible and extensible PPL, with a significant focus on ease of use, both for users and developers. It is built on top of the Julia programming language, which provides a high-performance environment for probabilistic modeling and inference, in addition to a rich ecosystem of packages for scientific computing and data analysis.

2.1 Basics

In `Turing.jl` a probabilistic model is defined in the form of a generative process, as described earlier. For example, the linear regression example (Example 1.1) can be implemented in `Turing.jl` as in Listing 1.

[2]Both in the statistical sense and the computational sense.

Here we define a standard Julia function `linear_regression`, which takes in the inputs x and defines the generative process outlined in (Linear regression). Then, the `@model` provided by `Turing.jl` transforms this function into a probabilistic model. As in the mathematical notation in (Linear regression), the syntax `left ~ right` is used to specify that a random variable `left` is distributed according to a distribution `right`. Note that all of the code inside the `@model` block is completely valid Julia code. The `@macro` performs minor transformations to the body of the `function`, by only touching the `~` operator and the `return` statement, making it easy to reason about the code and debug it. At a first glance, it might seem a bit strange that `x` is specified as an input to `linear_regression` but `y` is not; in `Turing.jl`, variables that are never to be considered random, i.e. they do not occur on the left-hand side of a `~` statement, need to be defined elsewhere in the scope of the model, be that as positional or keyword arguments to the model itself, variables in the body of the model, or just variables from the parent scope of the model, e.g. global variables. The full details of `@model` are described in Section 4.2.2.

```
using Turing                            # Load Turing.jl.

@model function linear_regression(x)    # Define the model using the `@model` macro.
    d = size(x, 1)                      # Get the dimension `d` from the input `x`.
    □² ~ InverseGamma(3, 4 / 10)        # Variance prior.
    □ ~ Normal(0, □10)                  # Intercept prior.
    □ ~ MvNormal(zeros(d), I)           # Coefficient prior.
    y ~ MvLogNormal(x' * □ .+ □, □² * I) # Likelihood.
end
```

Listing 1. Generative process for the linear regression model from Example 1.1 in `Turing.jl`.

With the joint `model` defined in Listing 1, we can `condition` the `model` on the variable y taking values as observed in a given dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, resulting in a model representing the posterior distribution $p(\sigma^2, \gamma, \beta|\mathcal{D}) \propto p(\mathcal{D}|\sigma^2, \gamma, \beta)p(\gamma, \beta \mid \sigma^2)p(\sigma^2)$. We can then perform inference on the model using one of the inference algorithms provided by `Turing.jl`, e.g. the No-U-Turn Sampler (NUTS) [7, 38] as in Listing 2:

```
model = linear_regression(x_train)          # Instantiate the model.
model_cond = model | (y = y_train,)         # Condition `model` on `y` equals `y_train`.
chain_nuts = sample(model_cond, NUTS(), 1000) # Perform 1000 MCMC iterations using NUTS.
chain_nuts_many = sample(model_cond, NUTS(), MCMCThreads(), 1000, 4) # 4 chains using threads.
```

Listing 2. Performing inference on the linear regression model from Listing 1.

We use the infix operator `|` to `condition` the `model` on the observed data `y`, and then call the `sample` function provided by `Turing.jl` to perform posterior inference. Note that `model | values` is just a shorthand for `condition(model, values)`. Further, `Turing.jl` also provides a `decondition` function to remove the conditioning on a variable.

With the default settings, each `sample` call in Listing 2 takes a few minutes to run on a laptop with Intel(R) Core(TM) i7-10710U CPU @ 1.10 GHz. We can also sample multiple chains in parallel, for example using multi-threading[3], as can be seen on the last line in Listing 2, resulting in multiple chains as seen in Figure 2.

Another similar operation one can perform on a `model` is `fix`, which is used to fix the value of a random variable to a specific value, but, unlike `condition`, the probability of the fixed value is not included in the model.

[3]This assumes that the Julia runtime has been started with multiple threads enabled.

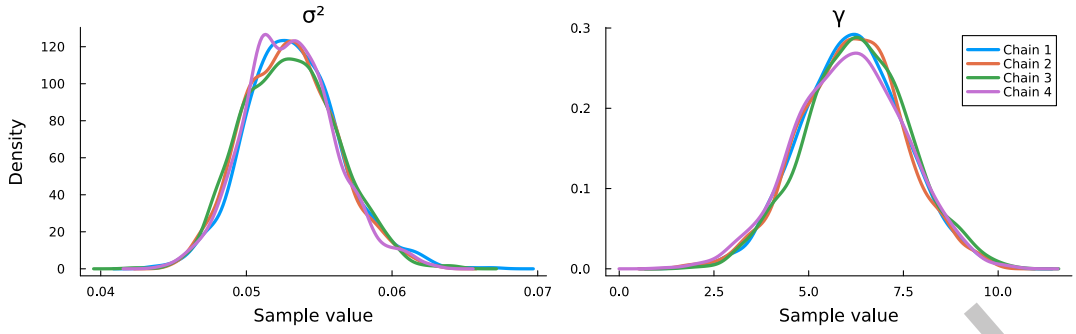

Fig. 2. Kernel density estimates (KDEs) of the density for σ^2 and γ obtained with the NUTS sampler for the linear regression model from Listing 1, as performed in Listing 2.

This can be viewed as a way of performing *intervention* [63] for a variable in the model and can be useful in many aspects of a standard Bayesian workflow [32]. For example, it is often useful in practice to verify that the inference algorithm of choice at least can recover the true parameters under the assumption that our model is not misspecified. Typically such a check is performed by generating synthetic data, in this case y, from the model for a specific parameter choice, and then verify that the resulting approximate posterior, in some sense, contain the true parameters used to generate the data. An example of this can be seen in Listing 3 whose results are visualized in Figure 3, where we can see that the true parameters for σ^2 and γ are indeed within the 95% credible intervals for the approximate posterior. Similar to `condition` having `decondition`, there is `unfix` to remove the fixing of variables in a model.

```
params_gen = (□² = 0.2, □ = 6, □ = randn(d) ./ □d) # Generate some parameters.
model_gen = fix(model, params_gen)                 # Fix `□²`, `□` and `□`.
(; y) = rand(model_gen)                            # Sample `y` from the `model_gen`.
chain_gen = sample(model | (y = y,), NUTS(), 1000) # Condition joint `model` on `y` and infer.
```

Listing 3. Generating data from the linear regression model from Listing 1 and performing inference.

For a more principled, but computationally more intensive approach to verify self-consistency, one can instead make use of *simulation-based calibration (SBC)* [80], which effectively repeats the above procedure with parameters sampled from the prior model and performs some hypothesis testing on the resulting estimates. This can be done easily in `Turing.jl` by using `rand` and `fix` in combination with the hypothesis testing tools available in Julia.

2.2 Inference and parameter identification

There are a wide range of inference algorithms available for models implemented in `Turing.jl`, both exposed directly to the user once they have executed the line `using Turing` and through external samplers provided by the Julia ecosystem that integrate nicely with `Turing.jl`. Some of the samplers available in `Turing.jl` are listed in Table 1.

Table 1 includes both the more automatic approaches, such as NUTS implemented in `Turing.jl` itself and external adaptive parallel tempering (PT) approaches implemented by Surjanovic et al. [76] in `Pigeons.jl`, which are particularly useful for users in (UG 1), and there are more specialized approaches such as combining, say, Metropolis-Hastings (MH) and elliptical slice sampling (ESS) in a Metropolis-within-Gibbs scheme to make use

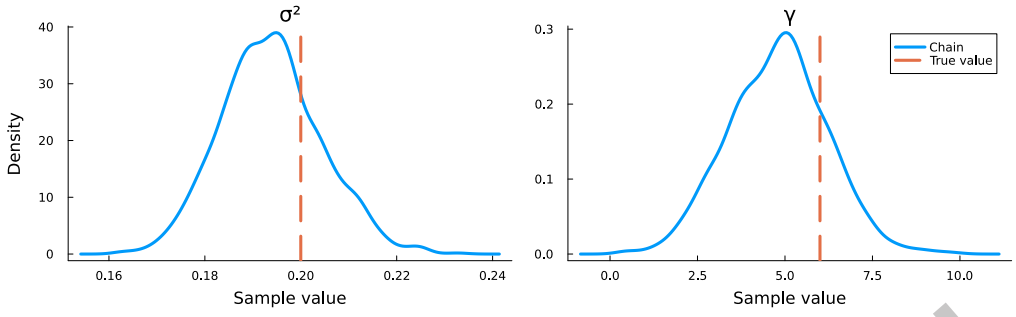

Fig. 3. Kernel density estimates (KDEs) of the density for σ^2 and γ obtained with the NUTS sampler for the linear regression model from Listing 1, as performed in Listing 3 using generated data with $\sigma^2 = 0.2$ and $\gamma = 6$, highlighted by the orange dashed lines.

Table 1. Some of the available samplers in `Turing.jl`

Category	Samplers	Julia Package
First class[a]	HMC and other variants, e.g. NUTS [38, 57]	`Turing.jl` + `AdvancedHMC.jl`[88]
	Metropolis-Hastings methods [37, 50]	`Turing.jl` + `AdvancedMH.jl`
	Affine invariant ensemble [30, 33]	`Turing.jl` + `AdvancedMH.jl`
	Standard Slice Sampling [56]	`Turing.jl` + `SliceSampling.jl`
	Elliptical Slice Sampling [54]	`Turing.jl` + `EllipticalSliceSampling.jl`
	Particle samplers e.g. Particle Gibbs[1]	`Turing.jl` + `AdvancedPS.jl`
	Metropolis-within-Gibbs	`Turing.jl`
Extensions[b]	Parallel tempering methods [78, 79]	`Pigeons.jl` [76] [c]
	Nested sampling [75]	`NestedSamplers.jl` [45]
	Pathfinder [89]	`Pathfinder.jl` [3] [c]
	HMC and other variants (alternative impl)	`DynamicHMC.jl` [60]

[a] These samplers are provided to the user upon `using Turing`. Note that they are mostly not implemented in `Turing.jl` itself, but rather in other packages maintained and developed by the same team, allowing users to benefit both within `Turing.jl` but also *outside* of `Turing.jl`.

[b] These samplers implement the `AbstractMCMC.jl` interface and so are trivially usable in `Turing.jl` by wrapping in `externalsampler(...)`, or they provide their own convenience method for `Turing.jl`.

[c] `Pigeons.jl` and `Pathfinder.jl` also support `Stan` through a bridging package in Julia and user-implemented log-density functions.

of the Gaussianity of parts of the prior distribution, which might be more of interest to users in (UG 2). As an example, we can also perform inference for Example 1.1 using a combination of MH and ESS as shown in Listing 4.

```
spl = Gibbs(
    @varname(□²) => MH(),
    @varname(□) => ESS(),
    @varname(□) => ESS()
)
chain_g = sample(model_cond, spl, MCMCThreads(), 1000, 4, discard_initial=1000, thinning=100)
```

Listing 4. Metropolis-within-Gibbs for the linear regression model in Example 1.1.

Note that in Listing 4, we have to use a much larger number of iterations to get a decent estimate of the posterior distribution; we're requesting 1000 samples in addition to discarding the initial 1000 samples (the burn-in) *and* thinning the chain by 100, which means we're running the sampler for 101,000 iterations in total. But the actual wall-clock time is in fact $\approx 2/3$ of the wall-clock time of using NUTS for all the parameters as was done in Listing 2. Hence there are scenarios where using a more specialized sampler can be beneficial, especially if the model involves non-differentiable computations, rendering gradient-based samplers like NUTS unusable. As we will see a bit later, we can in fact get much better computational performance than both Listing 2 and Listing 4 using NUTS with non-default settings.

Additionally, variational inference and optimization approaches such as maximum likelihood estimation (MLE) and maximum a posteriori (MAP) estimation are also available in `Turing.jl`, where the optimizers are provided by Julia packages like `Optim.jl` [51] and `Flux.jl` [40], through `Optimization.jl` [25]. As a result, in `Turing.jl` one has access to everything from gradient-free black box methods to higher-order gradient-based methods. For example, we can perform MAP estimation on the linear regression model from Listing 1 using L-BFGS from `Optim.jl` as shown in Listing 5.

```
using Optim                                  # Load Optim.jl.
result = optimize(model_cond, MAP(), LBFGS()) # Perform MAP estimation using L-BFGS.
```

Listing 5. Performing MAP estimation on the linear regression model from Listing 1 using Limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) from `Optim.jl`.

It's worth pointing out that a lot of this functionality provided by external packages is integrated with `Turing.jl` through what are called *package extensions* available in Julia since version 1.9; specifically, this means that the functionality bridging the external package and `Turing.jl` is *only loaded when the user explicitly calls for it*. This avoids increasing load and compile times for users who don't need that particular functionality, while still providing a seamless experience for those who do.

2.3 Post-inference analysis

Performing inference is generally just the first step in a Bayesian workflow; typically one wants to perform some kind of post-inference analysis to extract more useful information from the posterior samples than just the samples themselves. One also needs to check a number of diagnostics to ensure that the inference has been successful. In this section, we will go through some of the most common post-inference analysis tasks and how they can be performed in `Turing.jl`.

2.3.1 Convergence diagnostics. For the first class samplers listed in Table 1, the resulting `chain` object provides several diagnostics automatically. For example, Listing 6 shows the output of calling `summarystats` on the chain object for variables σ^2, γ, and a few of the β variables, obtained from the NUTS sampler in Listing 2:

```
Summary Statistics
  parameters      mean       std      mcse    ess_bulk    ess_tail      rhat   ess_per_sec
      Symbol   Float64   Float64   Float64     Float64     Float64   Float64       Float64

          □²    0.0543    0.0032    0.0000   5255.7222   2504.4390    1.0005       10.7193
           □    7.7252    0.3688    0.0066   3090.0819   2550.7065    1.0014        6.3024
        □[1]   -0.2270    0.1247    0.0022   3225.2858   2646.2490    1.0002        6.5781
        □[2]    0.0133    0.1229    0.0022   3252.0218   2688.1454    1.0007        6.6326
```

Listing 6. Output of calling `summarystats` on the chain obtained in Listing 2.

Here we immediately get an overview of some crucial statistics, such as the effective sample size (ESS) and the potential scale reduction factor ($\hat{R}$), which are both important for assessing the convergence of the sampler. ESS is a measure of how many independent samples the chain effectively contains, and the $\hat{R}$ is a measure of how well the chains have mixed. Moreover, in the above we see two versions of the ESS: one called `ess_bulk`, which is useful for assessing the convergence of the bulk of the distribution, and one called `ess_tail`, which is useful for assessing the convergence of the tails of the distribution. Roughly speaking, the ESS should be around the number of samples one is aiming for to perform the desired analysis (e.g. 1000), and $\hat{R}$ should be close to 1 for all parameters. We refer the reader to [87] for more details on these diagnostics.

2.3.2 Posterior predictive. As previously mentioned, one of the motivations for performing Bayesian inference might be to obtain predictions for new data with uncertainty quantification. In `Turing.jl`, making predictions is effectively just a matter of iterating over the samples present in a chain, fixing the joint model on said sample, and then sampling from the model. Since this is such a common operation, `Turing.jl` provides a simple function for it called `predict`:

```
model_train = linear_regression(x_train)               # Construct model with training inputs.
model_test = linear_regression(x_test)                 # Construct model with testing inputs.
chain_predict_train = predict(model_train, chain_nuts) # Predict for `model_train`.
chain_predict_test = predict(model_test, chain_nuts)   # Predict for `model_test`.
```

Listing 7. Obtaining the posterior predictive distribution for both training and testing data based on inference results in Listing 2.

Note that in Listing 7 we are creating new instances of the `linear_regression` model *without* conditioning the variables `y`. This is because we want to sample realizations of `y` conditioned on `chain_nuts`, hence we want a model which treats `y` as a random variable rather than a fixed value.

2.3.3 Other variables of interest. In some cases, one might want to inspect variables that are not directly present in the model as random variables, but are still of interest. In `Turing.jl` we do this through the **`return`** statement in the model block, which allows us to return any quantity we want as a post-inference step. For example, if we wanted to also extract the mean of the likelihood under the posterior, i.e. the expression `x' * □ .+ □` in the

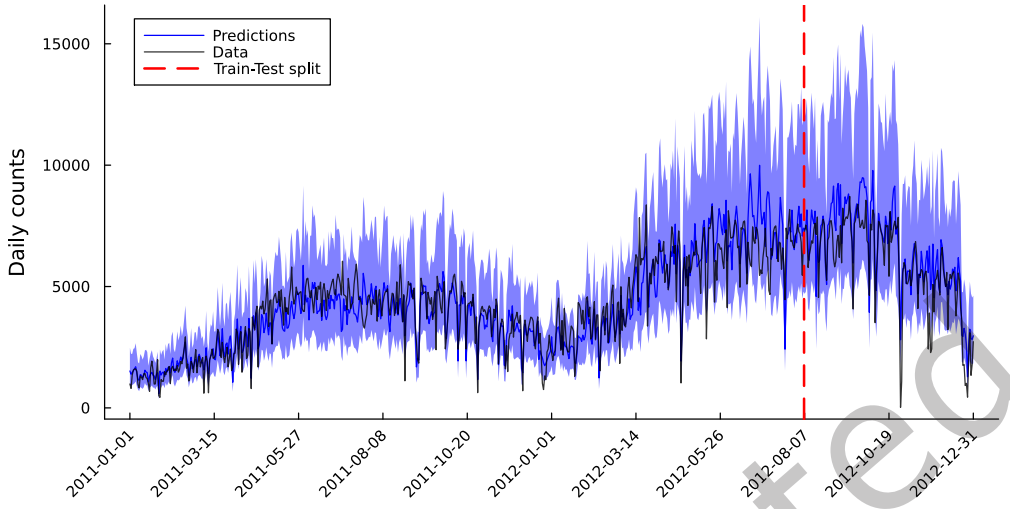

Fig. 4. Posterior predictive 95% quantiles for the bike-sharing model in Example 1.1 using the posterior samples from Listing 2.

linear regression model, we could simply add that to the `return` statement of the model and then call `returned` on the chain object in a way similar to `predict` from the previous section, as shown in Listing 8.

Note that in Listing 8 we are using a chain object obtained from the model `linear_regression` defined in Listing 1 to generate values for the slightly altered model `linear_regression_with_return` defined in Listing 8. This is a nice example of how the separation between the model and the inference object allows for easy reuse of the inference results in different contexts. We could have also just altered the `linear_regression` model directly; there is no issue with having a `return` statement in the model block even if we are performing inference.

```
# Returned quantites.
@model function linear_regression_with_return(x)
    d = size(x, 1)                                  # L2-L5 are the same.
    □² ~ InverseGamma(3, 4 / 10)
    □ ~ Normal(0, □10)
    □ ~ MvNormal(zeros(d), I)
    □ = x' * □ .+ □                                 # Linear predictor `□`.
    y ~ MvNormal(□, □²I)                            # Likelihood.
    return □                                        # Return `□`.
end

model_returned = linear_regression_with_return(x_data) # Instantiate model.
```

Listing 8. Extracting deterministic variables from the inference results in Listing 2.

2.4 Models within models

`Turing.jl` allows for models to be included within other models, allowing for a more modular approach to model building. There are several reasons why one might want to do this: break down a complex model into

smaller, more manageable components for easier debugging and testing, and reusing model components in other models, which may even be shared through a package system.

A prime example of the utility of this feature is the interoperability work done in [59] for pandemic response in the United Kingdom and internationally. In that work, the authors were working with a fairly large and complex model with many moving parts, and so it proved very useful to be able to break the model down into smaller, more manageable components, which could subsequently be combined in different ways to form the full model. This allowed the authors to test and debug the individual components in isolation and reuse the components in other models, enabling a much more efficient model development workflow.

```
@model function epimap(
    D,                          # Convolution weights for testing data.
    W,                          # Convolution weights for latent infections.
    F_id, F_out, F_in,          # Flux matrices used in regional flux model.
    K_time, K_spatial, K_local, # Kernel matrices used in the GP model.
    days_per_step = 1,          # Number of days per time step.
    num_cond = 30,              # Number of days used to initialize the process.
)
    # GP-model for R-value.
    R ~ to_submodel(SpatioTemporalGP(K_spatial, K_local, K_time))
    # Latent infections
    X ~ to_submodel(RegionalFlux(F_id, F_in, F_out, W, R; days_per_step))
    # Likelihood for testing data.
    C ~ to_submodel(NegBinomialWeeklyAdjustedTesting(X, D, num_cond))
end
```

Listing 9. Generative process for one of the models from [59] with annotations and a few minor modifications for the sake of readability. See [59] and [83] for more model details.

In Listing 9, we use `left ~ to_submodel(model)` syntax to include the models `SpatioTemporalGP`, `RegionalFlux`, and `NegBinomialWeeklyAdjustedTesting` in the "main" model `epimap`. `to_submodel` is used to indicate that in this particular case, the expression on the right of `~` is not a simple `Distribution`, but rather a model object whose **`return`** is to be sampled. A `Distribution` typically comes with both an implementation of `x = rand(dist)` and `logpdf(dist, x)`; a model whose **`return`** is what we are sampling does have a `rand`, but computing the log-density is generally intractable, as the return-values can be arbitrary expressions of both random and deterministic variables inside the `model`.

```
@model function SpatioTemporalGP(K_spatial, K_local, K_time)
    num_steps = size(K_time, 1)
    num_regions = size(K_spatial, 1)

    # Length scales.
    □_spatial ~ □□(0, 5)
    □_time ~ □□(0, 5)

    # Scales.
    □_spatial ~ □□(0, 0.5)
    □_local ~ □□(0, 0.5)

```

```
    # Gaussian Process using non-centered parameterization.
    E ~ filldist(Normal(), num_regions, num_steps)

    # Get Cholesky decomps using precomputed kernel matrices.
    L_space = spatial_L(K_spatial, K_local, □_spatial, □_local, □_spatial)
    U_time = time_U(K_time, □_time)

    # Obtain realization of log-R.
    log_R = L_space * E * U_time

    # Manually add the log-abs-det jacobian factor.
    Turing.@addlogprob! -kron_logabsdet(L_space, U_time)

    # Compute R.
    return exp.(log_R)
end
```

Listing 10. Efficient implementation of a spatio-temporal Gaussian Process (GP) using the non-centered parametrization, which is used in Listing 9. `spatial_L`, `time_U`, and `kron_logabsdet` are user-defined functions defined elsewhere. See [83] for more model details.

The "submodels" in Listing 9 are somewhat complicated, but a slightly simplified version of the `SpatioTemporalGP` model is shown in Listing 10. As we can see in Listing 10, `SpatioTemporalGP` is just another `Turing.jl` model, which returns the variable R that we need in `epimap`. See Teh et al. [83] and Nicholson et al. [59] for more details on these models.

Moreover, making use of the convenient packaging system in Julia, one can easily implement the `model` in a separate Julia package, which can then be shared with others, allowing for easy reuse of the model components both across different projects and different users.

An example of a package doing this is the `TuringGLM.jl` package which provides a collection of common regression models through a nice formula interface, heavily inspired by `brms` [15] and `bambi` [17]. As `TuringGLM.jl` uses `Turing.jl` models under the hood, these can then be embedded within other `Turing.jl` models in the same way as shown in Listing 9.

2.5 Integration with the Julia ecosystem

As already alluded to in Section 2.2, `Turing.jl` benefits greatly from the surrounding Julia ecosystem. In fact, `Turing.jl` itself is not a monolithic application but instead deeply integrated with the Julia ecosystem. This integration consists of two overlapping components:

- *Dependencies:* `Turing.jl` directly depends on popular Julia packages for much of its base functionality, such as `Distributions.jl` [6, 44] for definitions of and computations with probability distributions, and `ForwardDiff.jl` [68] for forward-mode automatic differentiation.
- *Utilities:* Beyond base functionality, users can reach for other Julia packages to either use directly in their models, such as `DifferentialEquations.jl` [66] for including differential equations in their model, or for other modeling-related tasks, such as data preparation with `DataFrames.jl` [12], or for downstream analysis of inference results using visualization packages `Plots.jl` [14, 19] and `Makie.jl` [23]. Most of these utilities "just work" due to the composable nature of the Julia ecosystem; for some packages the compatibility with `Turing.jl` is improved by additional optional features, such as autocorrelation plots of MCMC samples with `Plots.jl`.

This has a few benefits: a) users can leverage the full power of the Julia ecosystem in their models, b) users can easily integrate their own code with `Turing.jl` models, and c) the `Turing.jl` developers can focus on the core functionality related to probabilistic programming and MCMC, while relying on the Julia ecosystem for other functionality, such as data manipulation, plotting, automatic differentiation, and so on. This allows for a more modular and maintainable codebase, and ensures that `Turing.jl` can benefit from the rapid development of the Julia ecosystem.

In the following sections, we will discuss some of the most important components of the Julia ecosystem on which `Turing.jl` is based.

2.5.1 Distributions. Most of the probability distributions in `Turing.jl` available by default are implemented in `Distributions.jl` [6, 44]. In particular, `Distributions.jl` defines their (log) probability density functions, the functions to sample random variates from them, and their statistical properties such as mean and variance. `Turing.jl` supports every distribution that satisfies the `Distributions.jl` interface, including custom distributions defined by users.

2.5.2 Automatic differentiation (AD). Hamiltonian Monte Carlo methods such as NUTS require derivatives of a model's log density function. Fortunately, users do not have to calculate these derivatives by hand as `Turing.jl` can compute them using automatic differentiation (AD) [58]. In the Julia ecosystem, there are multiple packages for forward- and reverse-mode AD. Through the `LogDensityProblemsAD.jl` interface, `Turing.jl` supports the most widespread AD packages `ForwardDiff.jl` (forward-mode), `Zygote.jl` [39] (reverse-mode), `ReverseDiff.jl` (reverse-mode), and `Enzyme.jl` [52, 53] (forward and reverse mode). The `ForwardDiff.jl` package, arguably the most stable and best supported AD backend in Julia, is the default AD choice in Turing, as it is the most flexible AD backend available, supporting everything from mutation to custom data structures without issues, which not all AD packages can handle. However, users are encouraged to compare different AD alternatives with their model and switch to the AD backend that performs best, as this can vary drastically with dimension and workloads.

As an example, we return for a moment to the linear regression model implemented in Listing 1 for which we performed inference in Listing 2. As this is a somewhat midsized problem (hundreds of observations combined with almost 40 parameters) dominated by linear algebra operations, we might expect that reverse-mode AD would be the most efficient choice. Indeed, switching from `ForwardDiff.jl` to `ReverseDiff.jl`, which is a matter of an additional keyword argument to the `NUTS` constructor, results in a speed-up of roughly 10×.

2.5.3 Visualization. The two most popular plotting frameworks in Julia are `Plots.jl` [14, 19] and `Makie.jl` [23]. Both packages provide access to multiple graphics backends with a unified API. `Turing.jl` provides utilities for `Plots.jl` such as autocorrelation plots, trace plots, and density plots of MCMC chains.

2.5.4 Data. The `DataFrames.jl` [12] package is the most popular Julia package for working with tabular data. It is competitive in performance [12] compared to non-Julia alternatives such as `data.tables` and `polars`. The package is accompanied by an ecosystem of libraries for reading, writing, manipulating, and analyzing data. In the context of `Turing.jl`, `DataFrames.jl` can be used for data preparation of observations, which can be used in the probabilistic model in tabular format or in an array representation, analysis of inference runs, and more.

3 EXAMPLES

In this section we present a few examples of probabilistic models implemented in `Turing.jl` to illustrate some of the features discussed in the previous sections. Note that these examples are not in themselves novel, but are meant to illustrate some of the flexibility in models `Turing.jl` can handle, with a particular focus on the interoperability with the rest of the ecosystem.

3.1 Markov-Switching Generalized Autoregressive Conditional Heteroskedasticity (MS-GARCH)

In this example we follow [22] and consider a model for the volatility of the S&P 500 index using a *Markov-Switching Generalized Autoregressive Conditional Heteroskedasticity (MS-GARCH)* model. In financial econometrics, a quantity of interest is the *volatility* of an asset, which is typically measured as the standard deviation of the log-returns of the asset. This is often modeled as a function of the past returns using a *Generalized Autoregressive Conditional Heteroskedasticity (GARCH)* model [10], which is a time series model that captures the time-varying nature of volatility as follows:

$$\begin{aligned} \sigma_t^2 &:= \omega + \alpha x_{t-1}^2 + \beta \sigma_{t-1}^2 \quad \text{for } t = 2, \dots, N \\ x_t &\sim \mathcal{N}\left(0, \sigma_t\right) \quad \text{for } t = 2, \dots, N \end{aligned} \tag{GARCH}$$

where σ_i is the log-volatility at time t, x_i is the log-return at time t, and ω, α, and β are parameters of the model that we aim to infer. More accurately, the above is referred to as a GARCH(1, 1), indicating that we are using the first-order autoregressive and moving average terms for the volatility; in general a GARCH(p, q) model would use the p past squared returns and q past volatilities to model the current volatility. But a GARCH model assumes that the volatility is smoothly varying through time and thus cannot handle abrupt changes in volatility. Inspired by [22], we follow [36] and consider a *Markov-Switching GARCH (MS-GARCH)* model, where we assume that the volatility can switch between different regimes, each with its own GARCH model. In such a model, we hope to capture abrupt changes in volatility that are often observed in financial time series as a switch from one regime to another. An MS-GARCH model combines the GARCH model with a *Hidden Markov Model (HMM)* [5] to model the regime switching. Specifically, we now consider two separate GARCH(1, 1) models, one for each regime with separate parameters ω_k, α_k, and β_k for $k = 1, \dots, K$, and a random hidden state $z_i \in \{1, \dots, K\}$ that indicates which regime we are in at time t. Altogether, the model is given by the following generative process:

$$\begin{aligned} z_1 &\sim \text{Categorical}(\pi_0) \\ z_i &\sim \text{Categorical}(\pi[z_{i-1}, :]) \quad \text{for } t = 2, \dots, N \\ \sigma_{t,k}^2 &:= \omega_{z_i} + \alpha_k x_{t-1}^2 + \beta_k \sigma_{t-1}^2 \quad \text{for } k = 1, 2,\ t = 2, \dots, N \\ x_t &\sim \mathcal{N}\left(0, \sigma_{t,z_t}\right) \quad \text{for } t = 1, \dots, N \end{aligned} \tag{MS-GARCH}$$

where $(\omega_z, \alpha_z, \beta_z)_{z=1}^K$ are the GARCH parameters for the K different volatility regimes with constraints $\omega, \alpha, \beta > 0$, $\alpha < 1$ and $\beta < 1 - \alpha$, π_0 is the initial distribution, and π is the transition matrix of the hidden states with $\pi[z, :]$ denoting the z-th row of π, representing the transition probabilities from state z to all other states. The above assumes we have access to the GARCH parameters $(\omega_z, \alpha_z, \beta_z)_{z=1}^K$ and the transition matrix π; in practice, these are also parameters that we would need to infer from the data. Following [22], we let $K = 2$, with one level indicating "low volatility" and the other indicating "high volatility", and use the following priors for the parameters:

$$\begin{aligned} \pi_0 &:= [1/2, 1/2] \\ \pi[k, :] &\sim \text{Dirichlet}([1, 1]) \quad \text{for } k = 1, 2 \\ \omega &\sim \text{ordered}(\mathcal{N}_+(0, 0.25I)) \\ \alpha_k &\sim \mathcal{N}_{[0,1]}(0, 1) \quad \text{for } k = 1, 2 \\ \beta_k &\sim \mathcal{N}_{[0,1-\alpha_k]}(1, 1) \quad \text{for } k = 1, 2 \end{aligned} \tag{MS-GARCH prior}$$

where Dirichlet($[1, 1]$) is a symmetric Dirichlet distribution with concentration parameter 1 used to model the rows of the transition matrix π, ordered is a function that ensures that the elements of the multivariate variable ω are ordered ascendingly, representing the fact that we consider one state "low" volatility and the other "high" volatility, and finally $\mathcal{N}_{[a,b]}$ is a Normal distribution truncated to the interval $[a, b]$. The priors on α and β are

chosen such that a) $\alpha_k + \beta_k < 1$ to ensure that the volatility is mean-reverting, and b) β_k has more probability mass closer to 1 than α_k to ensure that the volatility is more influenced by the past volatility than the past squared returns.

Now, implementing this model in `Turing.jl` can be done in two ways. The first is to use the built-in support for discrete variables in `Turing.jl` and define the model exactly as in (MS-GARCH) with the discrete hidden states z_i explicitly included. This is show in Listing 22 in the appendix. We could then make use of the Gibbs sampler in `Turing.jl`, as briefly demonstrated in Listing 4, to perform inference jointly on the continuous variables using, say, HMC and discrete variables using, say, Particle Gibbs. This will work nicely for short time series, i.e. when working with a few discrete states, but will quickly become infeasible for longer time series or when the number of states is large.

This issue brings us to the second approach, which is to marginalize out the discrete variables and perform inference on the continuous variables only. In contrast to some other PPLs [55], `Turing.jl` does not currently provide automated ways to support this, and so the user has to do this by hand. Luckily, there is a package called `HiddenMarkovModels.jl` [21] which provides both efficient and differentiable implementations of typical HMM-related operations, such as filtering, smoothing, and marginalization. This allows us to define the generative process for the MS-GARCH model in `Turing.jl` but then use `HiddenMarkovModels.jl` to marginalize out the discrete variables, as shown in Listing 11.[4]

```
@model function ms_garch_marginalized(
    K,                                          # Number of states.
    T,                                          # Length of the time series.
    y,                                          # Time series observations.
)
    □ ~ filldist(Dirichlet(ones(K)), K)         # Transition matrix.
    □ ~ ordered(filldist(□□(0, 0.5), K))        # Initial volatility.
    □ ~ filldist(□□(0, 1, 1), K)                # Volatility persistence.
    □ ~ arraydist(□□.(1, 1, 1 .- □))            # Volatility persistence.

    □ = Matrix{Real}(undef, T, K)               # Volatilities.
    for k in 1:K                                # Initial volatilities.
        □[1, k] = □[k] / (1 - □[k] - □[k] + 1e-6)
    end
    for t = 2:T
        for k = 1:K                             # GARCH(1, 1) dynamics for volatilities.
            □[t, k] = □(□[k] + □[k] * y[t-1]^2 + □[k] * □[t-1, k]^2)
        end
    end

    □□ = fill(1/K, K)                           # Initial state probabilities
    trans = permutedims(□)                      # Transition matrix.
    emissions = eachrow(Normal.(0, □))          # Emission distributions.
    hmm = TimeHMM(□□, trans, emissions)         # Instantiate the custom HMM type.
    Turing.@addlogprob! logdensityof(hmm, y, 1:T) # Add the marginal log-likelihood.

    return (; □, □, y, y_pred=last(rand(hmm, 1:T)))
end
```

Listing 11. Generative process for the MS-GARCH model from Section 3.1 in `Turing.jl` with discrete parameters marginalized out using `HiddenMarkovModels.jl`.

[4]In practice, we need an additional argument to ensure good performance in for this model. See Listing 23 in the appendix for the full model.

Note that in Listing 11 the variable `y` is now *always* treated as an "observation" as it is passed as an explicit argument, allowing us to accumulate the corresponding log-likelihood by hand using the `Turing.@addlogprob!` macro. We can then use NUTS to perform inference on the continuous variables only, nicely identifying two volatility regimes in the data as shown in Figure 5, similarly to the results obtained in [22].

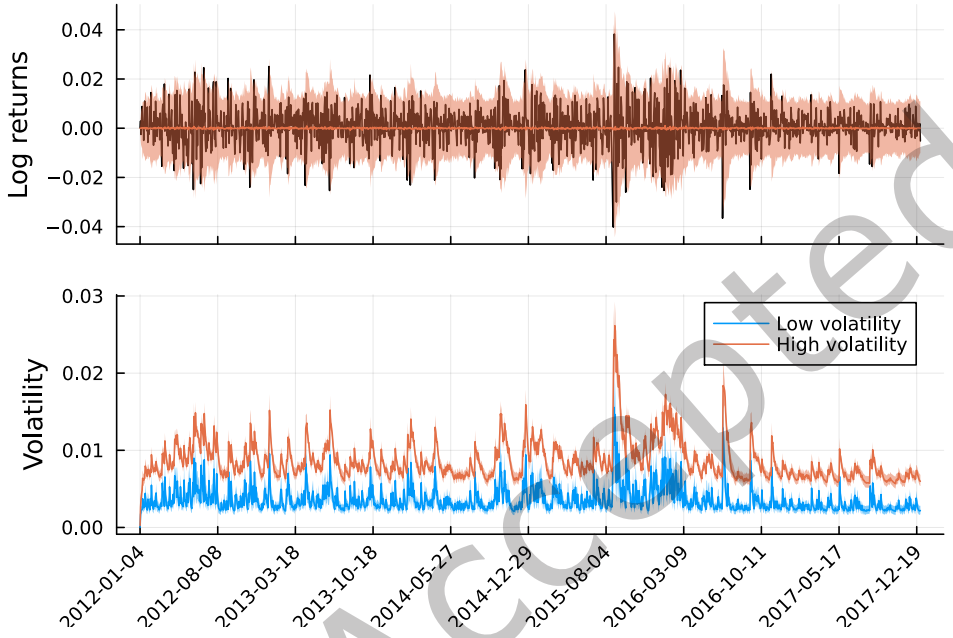

Fig. 5. Posterior predictive 95% quantiles and inferred volatility regimes for the MS-GARCH model from Section 3.1 using `Turing.jl` and `HiddenMarkovModels.jl`.

This also gives a more realistic demonstration of how `returned` can be useful for obtaining additional quantities from the model compared to the example in Section 2.3.3. To produce Figure 5, we used `returned` to extract the `y_pred` and `sigma` variables conditioned on the inference result.

3.2 Susceptible-Infectious-Recovered (SIR) model

Another area where probabilistic modeling has found great success is in the field of epidemiology [59, 83], where models are used to understand the spread of infectious diseases, both to predict future spread and to understand the underlying mechanisms of the spread. Bayesian inference is particularly well-suited for this task. Epidemiologists often have good ideas of the structure of the model, usually having written down a simulator of the disease spread given some parameters, in addition to having to deal with noisy observations due to the nature of how the data is acquired, e.g. through testing or self-reporting.

Here we follow [35] and model an outbreak of influenza A (H1N1) in a British boarding school from 1978 [2] using a simple Susceptible-Infectious-Recovered (SIR) model. A SIR model is a compartmental model used to model the spread of infectious diseases, where the population is divided into three compartments: susceptible (S), infectious (I), and recovered (R). The dynamics of the model are given by the following system of ordinary

differential equations:

$$\begin{aligned}\frac{dS}{dt} &= -\beta\frac{SI}{N}\\ \frac{dI}{dt} &= \beta\frac{SI}{N} - \gamma I\\ \frac{dR}{dt} &= \gamma I\end{aligned} \qquad \text{(SIR system)}$$

where β is the transmission rate, γ is the recovery rate, and $N = S + I + R$ is the total population size. In this school there were 763 male students and a total of 512 of them became ill during the 14-day outbreak. For each of these 14 days, the observations $(y_t)_{t=1}^{14}$ represent the number of students currently ill (specifically, the number of students in bed due to illness) on the given day t. Furthermore, on day 1 of the outbreak, only one student was ill, i.e. $I(0) = 1$, and the rest of the students were susceptible, i.e. $S(0) = 762$.

In this scenario, we have access to a sequence of observations $(y_t)_{t=1}^{14}$, and we want to infer the parameters β and γ of the SIR model that best explain the data. In [35], the authors suggest to use the following generative model for the data as a starting point:

$$\begin{aligned}\beta &\sim \mathcal{N}_+(2, 1)\\ \gamma &\sim \mathcal{N}_+(0.4, 0.5)\\ \phi^{-1} &\sim \text{Exponential}(0.2)\\ (S_t, I_t, R_t) &:= \text{solve}(\text{SIR}(\beta, \gamma), t) \quad \text{for } t = 1, \ldots, T\\ y_t &\sim \text{NegativeBinomial}(I_t, \phi) \quad \text{for } t = 1, \ldots, T\end{aligned} \qquad \text{(SIR Model)}$$

where $\mathcal{N}_+(\mu, \sigma)$ denotes a normal distribution with mean μ and standard deviation σ truncated from below at zero, $\text{Exponential}(\lambda)$ denotes an exponential distribution with rate λ, $\text{solve}(\text{SIR}(\beta, \gamma), t)$ denotes the solution of the SIR system (SIR system) at time t, and $\text{NegativeBinomial}(n, p)$ denotes a negative binomial distribution with mean n and dispersion parameter p. Finally, the likelihood is chosen as a negative binomial distribution with mean I_t and dispersion parameter ϕ to account for potential overdispersion in the data.

We can define and solve (SIR system) numerically using the `DifferentialEquations.jl` package [66], as seen in Listing 12.

```julia
using DifferentialEquations

const N = 763                                  # Population size.
function sir_ode!(du, u, p, t)
    S, I, R = u                                # Unpack the current state.
    □, □ = p                                   # Unpack the parameters.

    du[1] = -□ * I * S / N                     # dS
    du[2] = □ * I * S / N - □ * I              # dI
    du[3] = □ * I                              # dR
end

u0 = [N - 1, 1, 0.]                            # Initial: 1 infected, rest susceptible.
□ = 2.0                                        # Example transmission rate.
□ = 0.6                                        # Example recovery rate.
tspan = (0.0, 14.0)                            # Time span: 0 to 14 days.
prob = ODEProblem(sir_ode!, u0, tspan, [□, □]) # Instantiate the ODE problem.
```

Listing 12. Defining and solving the ODE (SIR system) for the SIR model from Section 3.2 using `DifferentialEquations.jl` [66].

```julia
@model function sir_model(problem)                    # `problem` is the SIR ODE system.
    □ ~ truncated(Normal(2, 1); lower=0)              # Transmission rate.
    □ ~ truncated(Normal(0.4, 0.5); lower=0)          # Recovery rate.
    □□¹ ~ Exponential(1/5)                            # Dispersion parameter.
    □ = inv(□□¹)

    problem_new = remake(problem, p=[□, □])           # Remake `problem` with new parameters.
    sol = solve(problem_new, saveat=1)                # Solve!
    if !issuccess(prob, sol)                          # Check if the solution was successful.
        Turing.@addlogprob! -Inf                      # If not, reject by adding `-Inf`.
        return nothing                                # Return early since we will reject.
    end
    sol_for_observed = sol[2, 2:end]                  # Infected people after the first day.
    n = length(sol_for_observed)                      # Number of observations.
    in_bed = Vector{Int}(undef, n)                    # Number of people in bed.
    for i = 1:n
        in_bed[i] ~ NegativeBinomial2(sol_for_observed[i] + 1e-5, □)
    end

    return (R□ = □/□, recovery_time = 1/□, sol = sol) # Return some quantities of interest.
end
```

Listing 13. Definition of the SIR model from (SIR Model) using `Turing.jl` and `DifferentialEquations.jl`.

Equipped with this, we can define the generative process for the SIR model in `Turing.jl` as shown in Listing 13, where we simply pass the `prob` object from Listing 12 to the model constructor in `Turing.jl`. Performing inference using NUTS, just as in Section 2.1, the resulting density and trace plots of the model parameters can be seen in Figure 6. In addition, making use of `returned` (Section 2.3), we compute the *reproduction number* R_0 and the *recovery time* of the disease, whose density plots are shown in Figure 7. Here we nicely recover the same values as in [35].

There are a few things in Listing 13 that are worth pointing out:

- Passing in an `ODEProblem` object to the model and calling `solve` "just works". In fact, no changes were made to `Turing.jl` or `DifferentialEquations.jl` to achieve this.
- We make use of a user-defined `issuccess(prob, sol)` function within the `model` to check if the ODE solver was successful or not, and if not, we add $-\infty$ to the log-joint accumulation causing the sampler to reject the sample. This is incredibly useful when working with models that are numerically unstable, in particular when using adaptive samplers such as `NUTS` where even parameter values unlikely under the posterior might be encountered during the adaptation phase.
- The sampler used, `NUTS`, is a gradient-based sampler, and so the above inference procedure implicitly makes use of the gradient computations *through* the `solve` call. Computing gradients through an ODE solver is a non-trivial problem with an extensive literature, but `DifferentialEquations.jl` automates a lot of this for us, with the additional option of choosing specific methods for the gradient computations through keyword arguments to the `solve` call. We refer the reader to [46] for a more detailed discussion of this topic. Similarly, we refer to [66] for more detailed discussions on the available solvers, their properties, and how to use them.

The ease of this integration, in addition to the performance achieved for these types of models, explain in part why `Turing.jl` with `DifferentialEquations.jl` has found several applications in recent years: epidemiology [41, 49], neuroscience [11, 73], biology [4, 81] and more [27, 71].

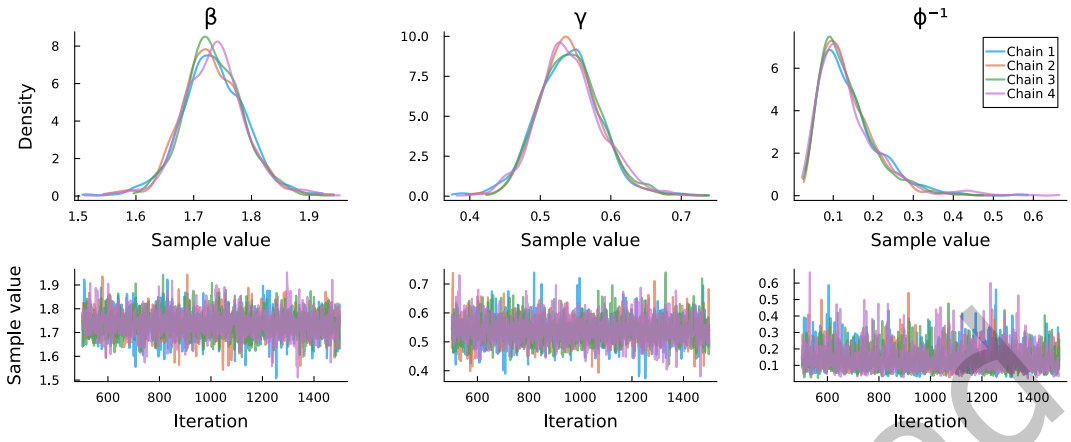

Fig. 6. Density and trace plots of the SIR model from Section 3.2 using `Turing.jl` and `DifferentialEquations.jl`.

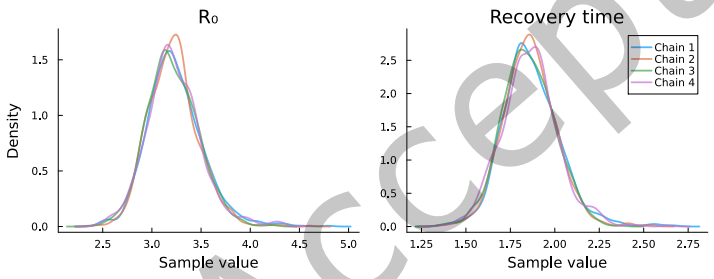

Fig. 7. Density plots for the reproducing number R_0 and recovery time of the SIR model from Section 3.2 using `Turing.jl` and `DifferentialEquations.jl`.

4 INTERNALS: MODELING SYNTAX AND INFERENCE IMPLEMENTATION DETAILS

In this section we will give an overview of the internals of `Turing.jl`, describing the main components of the library and how they interact with each other. This part in particular has changed drastically since the original inception of `Turing.jl` [31]. Today `Turing.jl` is effectively a shell around a number of other libraries, such as `DynamicPPL.jl` which implements the DSL of `Turing.jl`, `AbstractMCMC.jl` which defines the MCMC interface, `Bijectors.jl` [29] which handles the transformations of distributions from `Distributions.jl` [6], and so on. This modular design allows for a high degree of flexibility and extensibility, in addition to making the codebase more maintainable and easier to reason about, in particular in an open-source context where one wants to encourage contributions from the community. The fact that many of the packages are themselves useful *outside* of `Turing.jl` means that they often receive contributions from users who are not necessarily interested in nor using `Turing.jl`, which is a significant advantage when developing open-source software with limited resources.

We will start by giving a brief overview of crucial features of the Julia programming language that are leveraged extensively in `Turing.jl`, before moving on to describe the two main components of the library: `DynamicPPL.jl`, the DSL of `Turing.jl`, and `AbstractMCMC.jl` with its samplers.

4.1 Preliminaries on Julia

The Julia programming language [8] is a high-level, high-performance programming language for technical computing. It was designed to be both fast and approachable, with a syntax similar to widely used high-level languages such as Python and MATLAB. Julia is dynamically typed but performs just-in-time (JIT) compilation to native machine code using the LLVM compiler infrastructure [43], using an intermediate representation where types have, when possible, been inferred, allowing for efficient code generation. This means that Julia can be as fast as C or Fortran, while still being as easy to use as Python or MATLAB [8].

For an extensive introduction to Julia, we refer the reader to [8] and the official documentation. Here we will give a brief overview of some of the features of Julia that are particularly relevant to `Turing.jl`.

4.1.1 Types. Julia's type system is dynamic, allowing types to be determined at runtime, while simultaneously incorporating features more typical of static systems, such as explicit type annotations and parametric types, enabling generation of efficient code and, more importantly, inheritance through multiple dispatch. When types are not explicitly specified, any type is allowed. Types in Julia are organized hierarchically, with *abstract* types serving as supertypes for *concrete* types, which cannot themselves be subtyped. Moreover, only concrete types can be instantiated. This restriction prioritizes inheritance of behavior over inheritance of structure, addressing issues in traditional object-oriented programming (OOP) languages. All values in Julia are true objects with runtime-determined types, and variables are merely bindings to these values.

4.1.2 Multiple dispatch. Julia's type system supports polymorphism via multiple dispatch rather than, say, inheritance, enabling methods to specialize based on argument types, while the default assumption of unrestricted types allows gradual and optional adoption of explicit typing. This feature is central to the design of Julia itself, in addition to the ecosystem of packages built on top of Julia, including `Turing.jl`. A great example of this is the `+` operator, which is defined in the `Base` library as a function with many different method instances, each of which is optimized for different types of arguments. For example, a `SparseArray` will have a different specialized method instance for `+` than a standard dense `Array` containing the same elements.

4.1.3 Type inference and compilation. Julia leverages its type system to compile code to efficient native machine code using the LLVM infrastructure [43], enabling high performance while maintaining a high level of abstraction. There are two features that enable this: a) *method specialization*, where the compiler generates specialized versions of functions for different types of arguments, and b) *type inference*, where the compiler infers the types of variables and expressions in the code, allowing for further optimizations to be made, e.g. inlining of function calls.

In Julia, code is not fully compiled ahead-of-time like in some statically typed languages, but instead on an as-needed basis: the first time a function is called with a specific set of argument types, a (usually) specialized version of the function is generated and compiled. In this sense, Julia is a *just-in-time* (JIT) compiled language. However, if the types of the arguments cannot be inferred, the compiler will fall back to a slower, more generic version of the function.

Consider the following two functions, `f` and `g`, which both calculate the sum of two inputs:

```
f(x, y) = x + y
f(1, 2)::Int64
f(1.0, 2.0)::Float64

g(x, y) = x > 0 ? x + y : "x is not positive"
g(1, 2)::Union{Int64, String}
g(1.0, 2.0)::Union{Float64, String}
```

Listing 14. Example of two methods with identical behavior but differing type inference capabilities. The function calls are annotated with the inferred types, as given by `@code_warntype`.

This is of course a contrived example, but it illustrates the point: the first function, `f`, is *type stable*, in the sense that the return type of the function can be inferred to be a single concrete type from the input types, while the second function, `g`, is *type unstable*, as the return-type depends on a runtime check of the input values. As a result, these two functions result in completely different native code being generated: `f(1, 2)` results in a single native `add` instruction for the input type, while `g(1, 2)` results in a much longer and less efficient sequence of instructions.

4.1.4 Macros. Julia, taking inspiration from Lisp, represents its code as a data structure of the language itself called an *expression*. A `macro` is simply a function acting on such expressions: it takes an expression as input and returns another expression as output. This allows for powerful metaprogramming capabilities, as the macro can manipulate the expression in any way it sees fit before it is further processed by the Julia compiler. Note that this does mean that both the input expression and the output expression have to be valid Julia code, but the macro can do anything in between. This is a powerful feature that makes it trivial to define new DSLs within Julia, e.g. the DSL of `Turing.jl`.

4.2 `DynamicPPL.jl`: The modeling language

As mentioned, `DynamicPPL.jl` is the component of `Turing.jl` which is responsible for everything related to the domain-specific language (DSL) of `Turing.jl`. This is arguably the most complicated aspect of `Turing.jl`, as it is responsible for parsing the user's model, transforming it into a representation of a probabilistic model, and ensuring that the resulting model is efficient for the workloads of interest. There are a few key design principles that `DynamicPPL.jl` adheres to, closely following the rationale behind the design of Julia itself [8]:

(1) *Flexible by default, fast when needed:* `DynamicPPL.jl` attempts to support the definition of any probabilistic model imaginable, but clearly not all models can be compiled to efficient code. Hence `DynamicPPL.jl` effectively provides two models of operation: one backed by an *abstractly* typed trace structure, resulting in a slower but more flexible execution in terms of which types of models it can handle, and one backed by a *concretely* typed trace structure, resulting in improved performance but lower model flexibility. The mode `DynamicPPL.jl` uses depends on the actions that are being performed; for example, `DynamicPPL.jl` will usually execute the model once in the flexible mode to infer the types of the variables, and then switch to the performant mode for the actual inference.
(2) *Don't interfere with the user's code:* `DynamicPPL.jl` is designed to be as unobtrusive as possible. The user's code should look as much like regular Julia code as possible, with the DSL being used only where necessary. This is in contrast to some PPLs which require the user to write their entire model in the DSL, which can be quite cumbersome and unintuitive, in addition to making it harder to debug and reason about the code.
(3) *Don't reinvent the wheel:* Many PPLs will implement their own parser, compiler, and so on. `DynamicPPL.jl` takes the approach of making use of Julia's built-in parser and compiler as much as possible, delegating the heavy lifting to the Julia compiler. This has the advantage that `DynamicPPL.jl` can focus on the high-level aspects of the DSL, such as the syntax and semantics of the language, rather than on the low-level details of parsing and compiling. It also means lower complexity, less code to maintain, better integration with the rest of the Julia ecosystem, in addition to significantly lowered maintenance burden.

(1) and (2) in the above are very much geared towards (UG 1) and (UG 2), as it allows for a more flexible and intuitive user experience without too much magic happening to the user-provided code. (3) is more a design principle that benefits (UG 3) and (UG 4), as it allows for a more maintainable codebase and a more efficient use of resources, in addition to making it easier to integrate with the rest of the Julia ecosystem.

4.2.1 Syntax and semantics. As previously stated, the syntax and semantics of `DynamicPPL.jl`'s DSL, and thus `Turing.jl`, align with Julia in most cases, with only a few necessary deviations introduced by the DSL.

As seen in earlier examples, e.g. Listing 1, the `@model` macro is the entry point to the DSL of `DynamicPPL.jl`. `@model` takes in a single `function` expression as an argument and converts it into a probabilistic model, which is itself a Julia function. As a result, a model in `DynamicPPL.jl` follows the same syntax and semantics as any other Julia function, with the exception of a few minor alterations. The implementation details of `@model` will be explained in more detail in Section 4.2.2.

Within a `@model`, the key syntax is `left ~ right`, used to define random variables and observations. Here, `left` and `right` are Julia expressions, with the following constraints:

- `left` must be a simple expression, such as a symbol, property access, or indexed value (e.g., `a.b.c[1]`); expressions such as `a + b` are not allowed.
- `right` can be any valid expression that evaluates to a distribution-like object, e.g. `Normal()`.

Each `left ~ right` is interpreted as one of three instructions, based on these semantics:

- **`assume`**: `left` defines a random variable with `right` as its prior distribution.
- **`observe`**: `left` represents observed data with `right` as the likelihood.
- **`fix`**: `left` defines a deterministic variable, with `right` ignored.

The type of statement is determined by the following rules:

- **`assume`**: A variable is treated as an `assume` statement if it occurs on the left-hand-side of a `~` and is *not* considered an `observe` or `fix` statement.
- **`observe`**: A variable is treated as an `observe` statement if either of the following holds:
  - `left` has been explicitly conditioned by the user via `model | (left = ..., )` or `condition(model, left = ...)`.
  - `left` is passed as an argument to the model and `left !== missing` is `true`.
  - `left` is a literal value, e.g. `1.5` or `[1.0, 2.0]`.
- **`fix`**: A variable is treated as a `fix` statement if it occurs on the left-hand-side of a `~` and the user has explicitly fixed the variable via `fix(model, left = ...)`.

From the user's perspective, `left ~ right` is a single, unified syntax, dynamically parsed into one of the three instructions. This unification has several advantages: The `assume` and `observe` statements are essential for supporting diverse inference methods and are common in PPLs like `Venture` [47] and `Anglican` [85] (with varying terminology). They serve as flexible hooks for inference engines to locally modify model evaluation. The `fix` statement enables variable interventions by fixing their values, which could technically be handled by `assume`. However, a distinct `fix` improves clarity and allows for more efficient model evaluation. Unified `left ~ right` syntax simplifies code readability and enhances flexibility by enabling runtime determination of instructions, supporting composability in downstream tasks.[5]

Finally, it's worth noting that the `return` statement in a `@model` can be used just like in a standard Julia function. There can be one `return` statement, there can be multiple, they can occur within branches, and so on. However, the `return` statement can be useful for extracting expressions which are not part of `~` statements in a completely unrestricted manner, e.g. the full differential equation solution with all its auxiliary data, e.g. number of integrator steps taken, resulting from the `solve` call in Listing 13. This is used both to include models within models, as seen in Section 2.4, and in post-inference analysis using the `returned(model, chain)` function demonstrated in Section 2.3.

[5]Even in PPLs which require explicit declaration of assume and observe at model-definition-time it is possible to make the behavior adaptable by passing flags indicating the desired behavior for a given variable as an argument to the model. However, this quickly becomes tedious in many cases, e.g. when including models within models, one would have to thread such flags through to every submodel before reaching its intended target.

4.2.2 The `@model` macro. As mentioned briefly in the previous section, `@model` takes a user-defined `function` and transforms it into a model, which is a representation of the probabilistic model defined by the user. But, as stated earlier, `DynamicPPL.jl` wants to avoid too much interference with the user's code. Therefore, the `@model` macro really only does two things:

- **Defines methods `constructor` and `evaluator`.** Defines *two* methods from the one user-defined function: a `constructor` of the corresponding `Model` object, and an `evaluator` which is used to evaluate the constructed `Model`. This is illustrated in Listing 15 and Listing 16.
- **Conversion of `left ~ right`.** Replaces expressions of the form `left ~ right` with a conditional checking if `left` is a random variable, an observation (`condition`), or a deterministic variable (`fix`), and then executes different pipelines depending on the type of the left-hand side. This is illustrated in Figure 8.

Defines methods `constructor` and `evaluator`. To get a better understanding of the two methods generated by the `@model` macro, let us consider the following toy model:

```
1 @model function demo(args...; kwargs...)
2     x ~ Normal()
3     return "Hello, world!"
4 end
5 # Output: demo (generic function with 2 methods)
```

Listing 15. A toy model defined using the `@model` macro.

```
1 # Evaluator.
2 function demo(__model__, __varinfo__, __context__, args...; kwargs...)
3     ...
4 end
5 # Constructor.
6 demo(args...; kwargs...) = Model(demo, args, kwargs)
```

Listing 16. Pseudo-code for the `evaluator` and `constructor` resulting from Listing 15. See Listing 25 for an accurate annotated version. Note that type-annotations of the internal arguments `__model__`, `__varinfo__`, and `__context__` in the evaluator, which are crucial to ensure correct method dispatch, have been omitted for the sake of brevity.

Here we have a model which takes some arbitrary arguments `args...` and keyword arguments `kwargs...`, contains one random variable `x` which is distributed according to `Normal()`, and returns the string `"Hello, world!"`. Evaluating Listing 15 in the Julia REPL results in output seen on L5 in Listing 15, which indicates that the `@model` macro has generated two method instances for the function `demo`. As mentioned, one is the `constructor`, responsible for instantiating a `Model`, and the other is the `evaluator`, responsible for evaluating the model.

The `evaluator` is the main workhorse of the model and is where the transformed code provided by the user is embedded, while the `constructor` is fairly straightforward: it can be seen as a curried version of the `evaluator`, as it simply fills in user-provided arguments `args...` and keyword arguments `kwargs...` into the `evaluator`, leaving only the *internal* arguments used by `DynamicPPL.jl` to be filled in. As can be seen on L2 in Listing 16, there are three internal arguments provided to the `evaluator`:

(1) `__model__`: the model object that is the return value of the `constructor`. This might seem a bit redundant, but the resulting `Model` is then subsequently dispatched on to define custom behaviors for the model, such as how to sample from it, how to evaluate it, and so on.
(2) `__varinfo__`: a type that contains information about the variables in the model, such as their names, types, realizations, and so on. This takes the role of a *trace structure* as seen in other PPLs [48]. This trace structure is altered at every `left ~ right` statement, either mutably or immutably, through the corresponding pipeline functions `tilde_assume!!` and `tilde_observe!!`, as can be seen in Figure 8. See Section 4.2.4 for more on `__varinfo__`.
(3) `__context__`: a type specifying the context in which the model is evaluated, such as whether we are sampling from the prior, simply computing the log-probability, and so on. Similarly to `__varinfo__`, this is passed through the corresponding pipelines, allowing the contexts to alter the behavior of the model, as described in more detail in Section 4.2.5.

Thus, these internal arguments are available in the `evaluator`, and are subsequently passed to the pipeline handling the `~` statements. Note that, though not encouraged, the user can in fact access these objects inside the model, which can be useful for users in (UG 2) or (UG 4) who know what they are doing.

Conversion of `left ~ right`. As mentioned in Section 4.2.1, each expression in a `@model` of the form `left ~ right` is converted into one of the three types of instructions `assume`, `observe`, or `fix`. These instructions correspond to three different computational paths in the model evaluation. As a concrete example, consider the statement `x ~ Normal()` on L2 in Listing 15. A pseudocode representation of what `@model` generates for this statement is illustrated in Figure 8.

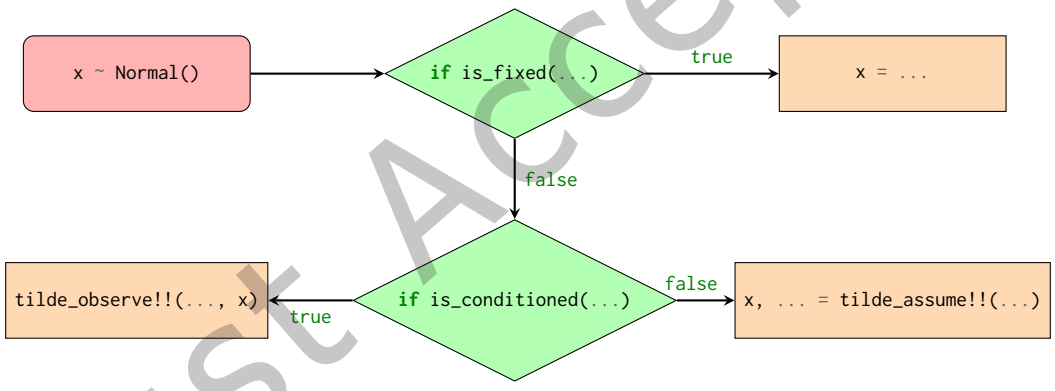

Fig. 8. Decision diagram for determining whether a variable should be considered fixed, assumed, or observed. The `~` statement being converted is represented by a red rectangle with rounded corners, decisions regarding whether variable `x` is considered fixed, observed or random, are represented by green diamond shapes, and the resulting computation path taken is represented by orange rectangles.

In Figure 8 we see there are three possible computation paths the variable `x` in the expression `x ~ Normal()`:

(1) `x` is a `fix` instruction, in which case the symbol `x` is assigned the specified value.
(2) `x` is an `observe` instruction, in which case we just call `tilde_observe!!(...)`.
(3) `x` is an `assume` instruction, in which case the symbol `x` is assigned the first return value of `tilde_assume!!(...)`, which corresponds to a realization of `x`.

As a result, the actual *code* modifications made by `@model` to the body of the user-specified **`function`** are very minor. That is, instead of modifying large chunks of the user-specified expression to implement the necessary evaluation

modes required in a PPL, `DynamicPPL.jl` (and by extension `Turing.jl`) instead modify the user-specified code's *behavior* through different method instances for `tilde_assume!!` and `tilde_observe!!`. See Section 4.2.5 for more details on this.

This approach has the benefit of giving the user a lot of flexibility in how they specify their models, as `@model` only interferes ever so slightly with the `~` statements, leaving the rest of the code untouched. However, this also means that `DynamicPPL.jl` assumes that the model is specified in topological order, i.e., that the dependencies of the variables are reflected in the ordering of the code, since if, say, a variable `y` is dependent on some other variable `x`, the variable `x` is only defined as a Julia variable after execution of its `~` statement.

4.2.3 The `VarName` *type.* As mentioned previously, in `DynamicPPL.jl`, we make use of dictionary-like structures to store information about variables in the model in `__varinfo__`, in `condition` and `fix`, etc. In these dictionary-like structures we use a special type of key, `VarName`, to represent the name of the variable being defined, providing additional information about the variable beyond what, say, a **`String`** or **`Symbol`** alone could provide. For example, a statement such as `x ~ Normal()` in a `@model` results in a **`VarName`**`{:x}()` embedded in the model body. The `VarName` is a representation of the *name* of the variable being defined, consisting of two things: the name of the variable as a **`Symbol`**, and a field containing any additional structure, e.g. indexing, `getproperty` calls, etc. This allows us to capture not only the name of the variable, but also how the variable is accessed in the model. The latter property is what allows usage of expressions on the left-hand side of the `~` statement that go beyond "simple" variable names. For example, Listing 17 a valid model definition:

```
@model function foo()
    y = Vector{Float64}(undef, 1)
    y[1] ~ Normal(0, 1)

    z = (x=nothing,)
    z.x ~ Normal(0, 1)
end
```

Listing 17. Example of model involving more complicated variable names.

As the `VarName` type can contain a lot of information obtained from the lowered Julia code, yielding itself poorly to manual construction, there exists a convenience macro `@varname` that can be used to construct a `VarName` object from a given Julia expression. For example, in the above example, we would end up with `@varname(y[1])`, containing the symbol `:y` and the indexing `_[1]`, and `@varname(z.x)`, containing the symbol `:x` and the `getproperty` call `_.x`.

4.2.4 Trace structure. As seen in earlier sections, a crucial aspect of the evaluation of a `Model` is the trace structure present in the `__varinfo__` variable in the evaluator. In short, this trace structure is a dictionary-like object with additional linearization capabilities to support conversion back and forth between a dictionary-like object and a linear sequence of values, e.g. an array. Moreover, these operations need to be performant when possible.

`DynamicPPL.jl` provides different implementations of the trace structure, each with their own pros and cons: some are mutable and some are immutable, some are optimized for specific workflows while others are optimized for generality, and so on.

However, one commonality between all trace structures is that they provide two modes of operation: a) a type *stable* mode, and b) a type *unstable* mode. When we talk about a trace structure being "type stable", we are not only referring to the fact that the trace structure itself is concretely typed rather than abstractly typed (see Section 4.1.1 for definitions of these terms), but *also* that other auxiliary information about a variable needed

during model evaluation infers to concrete types. To illustrate this, we will consider a simple example in which the trace structure cannot be concretely typed. We will use a `Dict` as an example trace structure, as it is sufficient to illustrate the issue.

```
@model function type_unstable_trace_1()
    x ~ Normal()
    y ~ Poisson()
end
```

Listing 18. Example of a model where ensuring that a `@model` is type stable can be challenging.

In Listing 18, the trace structure would have to hold two different types: for `x` we would need `Float64` and for `y` we would need `Int64`. This means that, using our example trace structure type, we would have to use a `Dict{VarName, Union{Float64, Int64}}` to store the realizations for the variables, which is not concretely typed due in part to the `Union{Float64,Int64}` element type. As a result, when we try to access the realization for, say, `x` in the model, the best Julia can do is to infer the type as `Union{Float64, Int64}`, and subsequent computations involving `x` would be type unstable. This is effectively the scenario we find ourselves in when using the type unstable mode in `DynamicPPL.jl`: in many ways, we get the same properties as if we used `Dict{VarName,Any}` to hold the realizations for the variables in the trace structure. Even if the realizations of the variables `x` and `y` were both `Float64`, e.g. `y ~ InverseGamma()` instead of `y ~ Poisson()`, types for auxiliary information about the variables, e.g. the transformation taking us from constrained to unconstrained space needed for some samplers such as standard HMC, might still end up being abstractly typed.

To address the aforementioned issue, the *type stable* mode in `DynamicPPL.jl` effectively creates a different trace structure for each unique symbol, *not* each unique `VarName`, occurring on the left-hand side of a `~` statement. Using `Dict` again as an example trace structure, instead of using a `Dict{VarName,Union{Float64,Int64}}` to store the realizations for both variables, we would use `NamedTuple{(:x, :y), Tuple{Float64,Int64}}`. For this, we can then implement a `getindex` that determines the correct type of `x` and `y` at *at compile time* instead of at runtime. This is possible because the symbols of the variables are type-parameters of `NamedTuple` *and* `VarName`, and so, making use of Julia's constant propagation and method specialization, the correct type for each variable is determined at compile time.

Not only do we group the realizations in this manner, we also group addition auxiliary data using the same approach. Grouping the trace structures by the symbol instead of the `VarName` itself is done for two reasons: a) the symbol of a variable in `@model` is static and so we can determine `VarName{sym}` at compile time, and b) it nicely corresponds to the semantic meaning users usually assign to variables in a model. To elaborate on the latter point, it is rare for a user to define a model with both `x[1] ~ Normal()` and `x[2] ~ Bernoulli()`; it is more natural to use a different symbol for these two `~` statements as they will usually carry different semantic meaning, e.g. `y ~ Normal()` and `z ~ Bernoulli()`. This is clearly a heuristic and not always the best way to segment the trace structure, but this turns out to be a great rule for the majority of models. [6] In practice, the typical case where multiple different `VarNames` share the same symbol is when they are used in `for` loops. We already saw an example of this on L16-L18 in Listing 13 where we end up with `n` `VarNames`, where each expression occurring on the left-hand side of a `~` shares the same symbol `:in_bed` and the distribution-type on the right-hand side of `~` is the same for all of these, which nicely fall within the cases handled with the aforementioned `NamedTuple`

[6] However, this is likely to change in the future: at the time of writing, `Turing.jl` has an experimental feature which makes use of abstract interpretation techniques to perform *static* analysis of the model to determine which trace structure is most appropriate. This is a very promising approach, but, as it is still experimental, we will not cover it in this report.

approach. And importantly, if the user does decide to write a model with such a structure, the model will still be evaluated correctly, but it will simply be computationally less efficient.

However, this leaves the problem of obtaining this typing information of the trace structure in the first place, as we don't necessarily want to put this burden on the user. For this, `DynamicPPL.jl` uses a simple heuristic:

(1) Execute model once in the type *unstable* mode to obtain variable names, their types, etc.
(2) Reify the type unstable trace structure into a type *stable* trace structure.

We can then use the type stable trace structure in the following computations, e.g. gradient computation for a HMC sampler, resulting in much more efficient machine code. There are of course cases where such an approach fails. For example, consider the following model:

```
@model function lazy_evaluation_insufficient()
    flip ~ Bernoulli(0.5)
    if flip
        x ~ Normal()
    else
        y ~ Poisson()
    end
end
```

Listing 19. Example of a model where a single execution of a model is insufficient to determine the types of the variables.

In Listing 19, which variable we see during a particular execution of the model depends on the realization of the variable `flip`: if `flip` evaluates to `true`, we never encounter the variable `y`, and vice versa for `x` if `flip` evaluates to `false`. As a result, simply evaluating this model a single time is insufficient to obtain a trace structure containing all the variables and their types. This does not mean that the above case cannot be handled at all; it only means that we are required to use a type unstable trace rather than the type stable trace, as the former is allowed to grow when new variables and types are encountered in the sampling process. And if high performance is crucial for the user's application, there is always the option of the user specifying a type stable trace themselves and passing this to the sampling process.

4.2.5 The tilde-pipeline and contexts. As alluded to in Figure 8, every `~` statement in a model will, unless it is considered `fixed`, hit a pipeline of functions that are responsible for updating the trace structure with the new information, computing log-probabilities, etc. Even though most inference methods absolutely do not need to touch these internals, e.g. `HMC` only really requires a map $x \mapsto \left(\log p(x), \nabla \log p(x)\right)$ which is provided elsewhere, it does enable full control over the inference process when needed. For example, particle MCMC methods implemented in `Turing.jl` makes use of continuation passing style to implement the resampling step in particle methods [31]. This is done by overloading the observe pipeline to yield back to the sampler transition step after each observation is processed, allowing communication between the particles, e.g. to resample particles based on the current state of the chain. For a detailed description of the pipeline, see Section B.1.2.

More typically, the pipeline is used to define various "modes of operation" for the model through the use of the `__context__` seen in Section 4.2.2. Inspired by contextual dispatch mechanisms in `Cassette.jl` [67] in addition to the effect handler approach of `Pyro` and `NumPyro` [9, 64], `__context__` is a stack-like structure of contexts that affect different stages of the call stack, unwrapping until a "leaf" / base context is reached. This makes it trivial to alter how a statement such as `left ~ right` is handled, even without making changes to the internals

of DynamicPPL.jl, potentially allowing users in (UG 2), (UG 3), and (UG 4) to implement custom behaviors. See Section B.1.3 for a detailed description of the context system and how it is used in DynamicPPL.jl.

One crucial aspect of this mechanism is that, since in most cases the contexts will have their types inferred by the Julia compiler, these different tilde_assume!! and tilde_observe!! calls can often be completely inlined, leading to zero or small runtime overhead. As a result, we can provide a very flexible way of altering the behavior of the model without sacrificing performance, which is crucial for a PPL that aims to be both user-friendly and high-performance.

4.3 AbstractMCMC.jl: The common MCMC interface

As previously mentioned, AbstractMCMC.jl is the package that defines the interface for MCMC algorithms in Turing.jl and beyond. AbstractMCMC.jl requires a sampler to implement two methods (Listing 20).

```julia
# Initial step function.
function AbstractMCMC.step(rng, model, sampler)
    # ...
    return initial_sample, initial_state
end

# Subsequent step function.
function AbstractMCMC.step(rng, model, sampler, state)
    # ...
    return sample, new_state
end
```

Listing 20. AbstractMCMC.jl's step interface

Here, the returned state is the next state in the Markov chain, which will subsequently be used as the input to the next call to step, and the returned sample is a, usually simpler, representation of the state that is consumable by the end-user, e.g. someone in (UG 1) or (UG 2). For example, it seems wholly redundant to pass the entire state of a HMC sampler, which includes the current joint energy of both the position and the momentum, potentially additional adaptation information, etc., to the end-user when all they really care about is the realizations of the random variables in their model. This is a very simple pattern to adhere to and nicely covers the vast majority of MCMC algorithms. Indeed, this pattern has been emerging as the go-to in other modern MCMC implementations; for example, in BlackJAX [16] a similar pattern is followed of having a single init method that returns the initial state, and then a step method that takes the current state and returns the next state.

Implementing these two methods comes with a host of benefits. For example, a user of the sampler can then simply call sample(model, sampler, num_iters) to perform MCMC inference on a model for a certain number of iterations. Further specialization can be done on the resulting sample-types to bundle the samples into a nicer representation, e.g. an MCMCChains.Chains instance from the MCMCChains.jl package. Moreover, if the sampler only requires log-density evaluations (and potentially gradients and higher-order derivatives), it can make use of the LogDensityProblems.jl [61] interface, thus becoming agnostic to the underlying model representation. In such a case, integration with Turing.jl is trivially done by wrapping the sampler in a call to externalsampler:

```
# Sampling from a Turing.jl `model` using a custom sampler `mysampler`.
chain = sample(model, externalsampler(my_sampler), num_iters)
```

Listing 21. Using a sampler, `mysampler` in this case, implementing `AbstractMCMC.jl`'s interface for `LogDensityProblems.jl` in `Turing.jl`.

All of the samplers listed in the "First class" category in Table 1 (with the exception of `AdvancedPS.jl`[7]) make use of this interface, thus enabling usage both within `Turing.jl` *and* outside of `Turing.jl`. This allows for a clean separation of concerns between the sampler and the rest of the inference process, and allows the sampler to be easily swapped out for another sampler that implements the same interface. In addition, it makes it easy for users in (UG 3) to implement their own samplers without having to worry about integration with PPLs such as `Turing.jl`, while still making it easy for users in (UG 1) and (UG 2) to make use of these samplers for their own models.

For the particularly interested, explicit examples of how to implement samplers can be found in Section B.2.

5 OTHER PROBABILISTIC PROGRAMMING LANGUAGES

As mentioned, a unique feature of `Turing.jl` is that it is entirely written in Julia; that is, the language that the user uses to define their probabilistic models is the same language that the `Turing.jl` developers use to implement the library. On the surface, this might not seem like a significant feature, but as we have seen in the previous sections, this has a number of important implications for the usability and extensibility of the library. Moreover, this feature is somewhat of a rarity in the world of PPLs.

Some of the PPLs that share this feature are `Anglican` [85] in Clojure, `Monad-Bayes` [74] in Haskell and `WebPPL` [34] in Javascript. However, these PPLs are implemented in programming languages that have less established ecosystems to support the kind of models that are of interest to the scientific and industry communities, who are the primary end-users of PPLs.

On the other hand, we have PPLs such as `Stan` [18], which define their own domain-specific language (DSL) for the specification of the model, which is subsequently transpiled into C++ code, in which model evaluation and inference is performed. To make the language more accessible to users, `Stan` also provides convenient interfaces in more user-friendly languages such as R, Python, and Julia. This approach has seen great empirical success, as `Stan` is one of the most widely used PPLs in the scientific community. However, the downside of this approach is that the user is limited to the features and capabilities of the DSL unless they are willing to dive into the internals of the `Stan` codebase and the world of C++. As a result, `Stan` caters really well to users in (UG 1) and somewhat to users in (UG 2), but historically not so much to users in (UG 3), as it requires additional expert knowledge of C++ to implement custom inference algorithms. This has improved significantly in recent times with the introduction of bridges to other programming languages such as Python and Julia [69], although one still suffers from the two-language problem of having to switch between the DSL and the host language.

`Nimble` [24] is another popular PPL that takes a similar approach: the user writes models in a DSL inspired by the `BUGS` language, which is then transpiled into C++ code. The compiled objects are then exposed via an interface in the R language [65], significantly lowering the barrier to entry for end-users in user groups (UG 1). Note that `Nimble` is quite different in the approach it takes to inference compared to `Stan`; `Stan` focuses on continuous variables with Hamiltonian Monte Carlo (HMC) [57] and its variants as the primary inference algorithms, while `Nimble` provides a more general-purpose inference engine which can handle both continuous and discrete

[7] SMC-based approaches require more structure in the model than a simple $x \mapsto \log p(x)$ map.

variables, and provides a wide range of inference algorithms such as particle MCMC[1], sequential Monte Carlo, and more.

Finally, we have PPLs which fall somewhere in between, such as `Pyro` [9], `NumPyro` [64], and `PyMC` [62, 72], which are both used and implemented in Python, but with the caveat that one has to adopt a certain underlying computational graph framework, such as PyTorch in the case of `Pyro`, Jax [13] in the case of `NumPyro`, and Theano [82] in the case of `PyMC3`[8]. This approach has the advantage that the user can leverage the full power of the underlying computational graph framework, which usually means significant performance improvements over more general-purpose approaches; however, it also means that the user is tied to a certain subset of the Python ecosystem that is also using the same computational graph framework. Generally speaking, implementing custom simulators in a way that is compatible with these PPLs requires some care, in particular for workloads that cannot be boiled down to standard linear algebra operations. Due to the popularity of these computational frameworks in the machine learning community, this is arguably less of a downside in the current landscape than it would have been a few years ago, but it is still a limitation users have to deal with.

Note that there are also other PPLs embedded in the Julia programming language. Probably the most well-known of these is `Gen.jl` [20], which is a PPL in Julia with a particular focus on the needs of user group (UG 3), allowing customization of the inference method to an extreme degree for whatever mode of interest. This has allowed them to achieve state-of-the-art performance on a number of interesting benchmarks [20, 70], however it also means that the PPL is arguably less accessible to users in (UG 1) and even (UG 2).

Overall, `Turing.jl` stands out not because of any singular feature—such as its syntax, supported inference algorithms, or model expressiveness—but through the combination of these attributes within a high-level yet performant language. Like other PPLs, `Turing.jl` adopts a convenient `left ~ right` syntax, e.g. as in `BUGS`, `Stan`, and `Nimble`, `assume` / `observe` primitives, e.g. as in `Venture`, `Anglican`, and `Church`, and automatic differentiation for gradient-based samplers, e.g. as in `Stan`, `PyMC`, `Pyro`, and `NumPyro`). It also borrows some contextual evaluation ideas from `Pyro` and `NumPyro`.

What sets `Turing.jl` apart is its nonintrusive model declaration, seamless integration with Julia's scientific computing ecosystem, and computational efficiency leveraging Julia's compiler infrastructure. These strengths make it attractive to both basic users (UG 1) seeking simplicity and generality, and more advanced users, e.g. (UG 2) and (UG 3), who value performance and extensibility for custom simulators and inference algorithms.

However, this flexibility comes with trade-offs. `Turing.jl` imposes minimal structure, allowing users to "shoot themselves in the foot" if not cautious. Although runtime checks and documentation aim to mitigate misuse, this can challenge beginners compared to structured PPLs like `Stan` or `Nimble`, which reduce errors through enforced patterns. Additionally, performance in `Turing.jl` depends on efficient model implementation, which may be difficult for users unfamiliar with Julia's performance characteristics. In contrast, frameworks such as `NumPyro` and `PyMC` limit users to subsets of their host languages, resulting in more predictable performance and fewer pitfalls at the cost of flexibility.

6 CONCLUDING REMARKS

Probabilistic programming is becoming increasingly important in many fields, including economics, epidemiology, biology, and Bayesian statistics research. The key idea of probabilistic programming is to decouple inference and model specification. This allows modeling practitioners to design and optimize their model for a particular task of interest without being distracted by inference details. On the other hand, it encourages inference researchers to create more general-purpose inference algorithms that are efficient on a broader range of models.

[8] `PyMC` is the successor to `PyMC3`, which uses `PyTensor` as its computational backend, allowing the user to choose between code generation for C, Jax, or Numba [42].

Furthermore, models specified in probabilistic programming languages are often more concise and modular than manually implemented models thanks to the abstraction away from inference implementation details. This increased readability and modularity supports better communication of statistical models, more thorough model checking, and reproducible research.

Many PPLs have been developed with design trade-offs between performance, flexibility, and target user groups. This article introduces the `Turing.jl` language. `Turing.jl` has an intuitive syntax for modeling and provides a wide range of state-of-the-art inference options. `Turing.jl` is designed to be inference algorithm agnostic, in the sense that it does not impose a particular inference algorithm upon the user. Users can use the default option or select an algorithm based on the model. `Turing.jl` also provides advanced functionalities for working with models, e.g. conditioning/deconditioning, fixing variables, making predictions, computing generated quantities, and using models as sub-models for better modularity. Furthermore, `Turing.jl` models have full interoperability and composability with external Julia libraries, which allows users to benefit from an entire ecosystem of scientific computing and modeling: to name a few, `DifferentialEquations.jl` for mechanistic modeling through differential equations, `DataFrames.jl` for working with tabular data, `Distributions.jl` for an extensive range of probability distributions, and `ForwardDiff.jl` and `ReverseDiff.jl` among other options for automatic differentiation. In addition, the entire `Turing.jl` language is implemented in the Julia language; thanks to Julia's high performance and concise syntax, `Turing.jl` is friendly across a broader range of user groups from beginners to inference researchers to PPL developers.

7 ACKNOWLEDGEMENTS

This work was partially funded by the Alan Turing Institute. SA is a member of the Cluster of Excellence Machine Learning: New Perspectives for Science, University of Tübingen and is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC number 2064/1 – Project number 390727645. MT acknowledges funding from the Research Council of Finland (grant number 347279). TF and HG acknowledge funding from Huawei R&D (UK). HG and ZG acknowledge funding from EPSRC under grant EP/W002965/1.

Since Turing language is an open-source project, we have received many contributions from volunteers around the world. We would like to thank all contributors for their valuable work.

REFERENCES

[1] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. 2010. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72, 3 (2010), 269–342.

[2] Anonymous. 1978. Influenza in a boarding school. *British Medical Journal* 1 (mar 1978), 587.

[3] Seth Axen. 2021. *Pathfinder.jl.* Zenodo. https://doi.org/10.5281/ZENODO.5914976

[4] Marcelo Batista, Fabio Montoni, Cristiano Campos, Ronaldo Nogueira, Hugo A Armelin, and Marcelo S Reis. 2023. A Framework for Inference and Selection of Cell Signaling Pathway Dynamic Models. In *Brazilian Symposium on Bioinformatics*. Springer, 82–93.

[5] Leonard E Baum and Ted Petrie. 1966. Statistical inference for probabilistic functions of finite state Markov chains. *The annals of mathematical statistics* 37, 6 (1966), 1554–1563.

[6] Mathieu Besançon, Theodore Papamarkou, David Anthoff, Alex Arslan, Simon Byrne, Dahua Lin, and John Pearson. 2021. Distributions.jl: Definition and Modeling of Probability Distributions in the JuliaStats Ecosystem. *Journal of Statistical Software* 98, 16 (2021), 1–30. https://doi.org/10.18637/jss.v098.i16

[7] Michael Betancourt. 2017. A Conceptual Introduction To Hamiltonian Monte Carlo. *CoRR* (2017). arXiv:1701.02434 [stat.ME] http://arxiv.org/abs/1701.02434v2

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. 2017. Julia: A fresh approach to numerical computing. *SIAM review* 59, 1 (2017), 65–98.

[9] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of machine learning research* 20, 28 (2019), 1–6.

[10] Tim Bollerslev. 1986. Generalized autoregressive conditional heteroskedasticity. *Journal of econometrics* 31, 3 (1986), 307–327.

[11] Peter A. R. Bork, Natalie L. Hauglund, Yuki Mori, Kjeld Møllgård, Poul G. Hjorth, and Maiken Nedergaard. 2024. Modeling of Brain Efflux: Constraints of Brain Surfaces. *Proceedings of the National Academy of Sciences* 121, 16 (2024). https://doi.org/10.1073/pnas.2318444121
[12] Milan Bouchet-Valat and Bogumił Kamiński. 2023. DataFrames.jl: Flexible and Fast Tabular Data in Julia. *Journal of Statistical Software* 107, 4 (2023), 1–32. https://doi.org/10.18637/jss.v107.i04
[13] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax
[14] Tom Breloff. 2024. *Plots.jl*. https://doi.org/10.5281/zenodo.10959005
[15] Paul-Christian Bürkner. 2021. Bayesian Item Response Modeling in R With brms and Stan. *Journal of Statistical Software* 100, 5 (2021). https://doi.org/10.18637/jss.v100.i05
[16] Alberto Cabezas, Adrien Corenflos, Junpeng Lao, Rémi Louf, Antoine Carnec, Kaustubh Chaudhari, Reuben Cohn-Gordon, Jeremie Coullon, Wei Deng, Sam Duffield, Gerardo Durán-Martín, Marcin Elantkowski, Dan Foreman-Mackey, Michele Gregori, Carlos Iguaran, Ravin Kumar, Martin Lysy, Kevin Murphy, Juan Camilo Orduz, Karm Patel, Xi Wang, and Rob Zinkov. 2024. Blackjax: Composable Bayesian Inference in Jax. *CoRR* (2024). arXiv:2402.10797 [cs.MS] http://arxiv.org/abs/2402.10797v2
[17] Tomás Capretto, Camen Piho, Ravin Kumar, Jacob Westfall, Tal Yarkoni, and Osvaldo A Martin. 2022. Bambi: A Simple Interface for Fitting Bayesian Linear Models in Python. *Journal of Statistical Software* 103, 15 (2022), 1–29. https://doi.org/10.18637/jss.v103.i15
[18] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of statistical software* 76 (2017).
[19] Simon Christ, Daniel Schwabeneder, Christopher Rackauckas, Michael Krabbe Borregaard, and Thomas Breloff. 2023. Plots.jl – a user extendable plotting API for the Julia programming language. (2023). https://doi.org/10.5334/jors.431
[20] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. 2019. Gen: A General-purpose Probabilistic Programming System with Programmable Inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 221–236. https://doi.org/10.1145/3314221.3314642
[21] Guillaume Dalle. 2024. *HiddenMarkovModels.jl: generic, fast and reliable state space modeling*. Zenodo. https://doi.org/10.5281/ZENODO.10931812
[22] Luis Damiano, Brian Peterson, and Michael Weylandt. 2024. A Tutorial on Hidden Markov Models using Stan. *GitHub* (April 2024). https://github.com/luisdamiano/stancon18 [Online; accessed 30. Apr. 2024].
[23] Simon Danisch and Julius Krumbiegel. 2021. Makie.jl: Flexible high-performance data visualization for Julia. *Journal of Open Source Software* 6, 65 (2021), 3349. https://doi.org/10.21105/joss.03349
[24] P. de Valpine, D. Turek, C.J. Paciorek, C. Anderson-Bergman, D. Temple Lang, and R. Bodik. 2017. Programming with models: writing statistical algorithms for general model structures with NIMBLE. *Journal of Computational and Graphical Statistics* 26 (2017), 403–417. https://doi.org/10.1080/10618600.2016.1172487
[25] Vaibhav Kumar Dixit and Christopher Rackauckas. 2023. *Optimization.jl: A Unified Optimization Package*. Zenodo. https://doi.org/10.5281/ZENODO.7738525
[26] David J Earl and Michael W Deem. 2005. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics* 7, 23 (2005), 3910–3916.
[27] Ahmed Elmokadem, Yi Zhang, Timothy Knab, Eric Jordie, and William R Gillespie. 2023. Bayesian pbpk modeling using r/stan/torsten and julia/sciml/turing. jl. *CPT: Pharmacometrics & Systems Pharmacology* 12, 3 (2023), 300–310.
[28] Hadi Fanaee-T. 2013. Bike Sharing. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5W894.
[29] Tor Erlend Fjelde, Kai Xu, Mohamed Tarek, Sharan Yalburgi, and Hong Ge. 2020. Bijectors. jl: Flexible transformations for probability distributions. In *Symposium on Advances in Approximate Bayesian Inference*. PMLR, 1–17.
[30] Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman. 2013. <tt>emcee</tt>: the Mcmc Hammer. *Publications of the Astronomical Society of the Pacific* 125, 925 (2013), 306–312. https://doi.org/10.1086/670067
[31] Hong Ge, Kai Xu, and Zoubin Ghahramani. 2018. Turing: A Language for Flexible Probabilistic Inference. In *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 84)*, Amos Storkey and Fernando Perez-Cruz (Eds.). PMLR, 1682–1690. https://proceedings.mlr.press/v84/ge18b.html
[32] Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. 2020. Bayesian Workflow. *CoRR* (2020). arXiv:2011.01808 [stat.ME] http://arxiv.org/abs/2011.01808v1
[33] Jonathan Goodman and Jonathan Weare. 2010. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science* 5, 1 (2010), 65–80.
[34] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2024-11-13.
[35] Léo Grinsztajn, Elizaveta Semenova, Charles C Margossian, and Julien Riou. 2021. Bayesian workflow for disease transmission modeling in Stan. *Statistics in medicine* 40, 27 (2021), 6209–6234.

[36] M. Haas. 2004. A New Approach To Markov-Switching Garch Models. *Journal of Financial Econometrics* 2, 4 (2004), 493–530. https://doi.org/10.1093/jjfinec/nbh020
[37] W Keith Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. (1970).
[38] Matthew D. Hoffman and Andrew Gelman. 2011. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *CoRR* (2011). arXiv:1111.4246 [stat.CO] http://arxiv.org/abs/1111.4246v1
[39] Michael Innes. 2018. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *CoRR* abs/1810.07951 (2018). arXiv:1810.07951 http://arxiv.org/abs/1810.07951
[40] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. 2018. Fashionable Modelling with Flux. *CoRR* abs/1811.01457 (2018). arXiv:1811.01457 https://arxiv.org/abs/1811.01457
[41] Fabienne Krauer, Tor Erlend Fjelde, Mihaly Koltai, David Hodgson, Marina Treskova-Schwarzbach, Christine Harvey, Mark Jit, Ole Wichmann, Thomas Harder, and Stefan Flasche. 2022. Estimating RSV seasonality from pandemic disruptions: a modelling study. *medRxiv* (2022).
[42] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. 1–6.
[43] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.
[44] Dahua Lin, David Widmann, Simon Byrne, John Myles White, Andreas Noack, Mathieu Besançon, Douglas Bates, John Pearson, John Zito, Alex Arslan, Moritz Schauer, Kevin Squire, David Anthoff, Theodore Papamarkou, Jan Drugowitsch, Benjamin Deonovic, Avik Sengupta, Seth Axen, Brian J Smith, Giuseppe Ragusa, Glenn Moynihan, Christoph Dann, Mike J Innes, Mohamed Tarek, Michael, Martin O'Leary, Gustavo Lacerda, and Iain Dunning. 2024. *JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions.* https://doi.org/10.5281/zenodo.11001059
[45] Miles Lucas, , Tor Erlend Fjelde, Alexander Plavin, Hong Ge, and Cameron Pfiffer. 2023. *TuringLang/NestedSamplers.jl: v0.8.3.* Zenodo. https://doi.org/10.5281/ZENODO.8009875
[46] Yingbo Ma, Vaibhav Dixit, Mike Innes, Xingjian Guo, and Christopher Rackauckas. 2018. A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions. *CoRR* (2018). arXiv:1812.01892 [cs.NA] http://arxiv.org/abs/1812.01892v2
[47] Vikash K Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. 2018. Probabilistic programming with programmable inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 603–616.
[48] Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. 2018. An Introduction To Probabilistic Programming. *CoRR* (2018). arXiv:1809.10756 [stat.ML] http://arxiv.org/abs/1809.10756v1
[49] Gustavo S Mesch, Wilson Levy Braga da Silva Neto, and Jose Eduardo Storopoli. 2022. Media exposure and adoption of COVID-19 preventive behaviors in Brazil. *New Media & Society* (2022), 14614448221122203.
[50] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. 1953. Equation of state calculations by fast computing machines. *The journal of chemical physics* 21, 6 (1953), 1087–1092.
[51] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. 2018. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software* 3, 24 (2018), 615. https://doi.org/10.21105/joss.00615
[52] William Moses and Valentin Churavy. 2020. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12472–12485. https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf
[53] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165
[54] Iain Murray, Ryan Prescott Adams, and David JC MacKay. 2010. Elliptical slice sampling. (2010).
[55] Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43.
[56] Radford M Neal. 2003. Slice sampling. *The annals of statistics* 31, 3 (2003), 705–767.
[57] Radford M. Neal. 2012. Mcmc Using Hamiltonian Dynamics. *CoRR* (2012). arXiv:1206.1901 [stat.CO] http://arxiv.org/abs/1206.1901v1
[58] Richard D. Neidinger. 2010. Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming. *SIAM Rev.* 52, 3 (2010), 545–563. https://doi.org/10.1137/080743627 arXiv:https://doi.org/10.1137/080743627
[59] George Nicholson, Marta Blangiardo, Mark Briers, Peter J Diggle, Tor Erlend Fjelde, Hong Ge, Robert JB Goudie, Radka Jersakova, Ruairidh E King, Brieuc CL Lehmann, et al. 2022. Interoperability of statistical models in pandemic preparedness: principles and reality. *Statist. Sci.* 37, 2 (2022), 183–206.

[60] Tamas K. Papp, David Widmann, Dilum Aluthge, Yimin Yi, delehef, Julia TagBot, Morten Piibeleht, Pietro Monticone, and st. 2023. *tpapp/DynamicHMC.jl: v3.4.7*. Zenodo. https://doi.org/10.5281/ZENODO.10225136

[61] Tamas K. Papp, David Widmann, Christopher Rackauckas, Tor Erlend Fjelde, Andreas Scheidegger, Dilum Aluthge, Julia TagBot, Stefan Karpinski, andreaskoher, and trahflow. 2024. *tpapp/LogDensityProblems.jl: v2.1.2*. Zenodo. nil pages. https://doi.org/10.5281/ZENODO.13885021

[62] Anand Patil, David Huard, and Christopher J Fonnesbeck. 2010. PyMC: Bayesian stochastic modelling in Python. *Journal of statistical software* 35, 4 (2010), 1.

[63] Judea Pearl. 2009. *Causality*. Cambridge university press.

[64] Du Phan, Neeraj Pradhan, and Martin Jankowiak. 2019. Composable effects for flexible and accelerated probabilistic programming in NumPyro. *arXiv preprint arXiv:1912.11554* (2019).

[65] R Core Team. 2021. R: A Language and Environment for Statistical Computing. https://www.R-project.org/

[66] Christopher Rackauckas and Qing Nie. 2017. DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software* 5, 1 (2017).

[67] Jarrett Revels, Valentin Churavy, Shuhei Kadowaki, Simeon David Schaub, Tim Besard, Frames White, Mike J Innes, Twan Koolen, Nathan Daly, Kristoffer Carlsson, Yuto Horikawa, Xiu zhe (Roger) Luo, Robin Deits, Oscar Smith, Morten Piibeleht, Moritz Schauer, Keno Fischer, Dilum Aluthge, Chris de Graaf, and Avik Pal. 2024. *JuliaLabs/Cassette.jl: v0.3.13*. Zenodo. https://doi.org/10.5281/ZENODO.10529466

[68] J. Revels, M. Lubin, and T. Papamarkou. 2016. Forward-Mode Automatic Differentiation in Julia. *arXiv:1607.07892 [cs.MS]* (2016). https://arxiv.org/abs/1607.07892

[69] Edward A. Roualdes, Brian Ward, Bob Carpenter, Adrian Seyboldt, and Seth D. Axen. 2023. BridgeStan: Efficient in-memory access to the methods of a Stan model. *Journal of Open Source Software* 8, 87 (2023), 5236. https://doi.org/10.21105/joss.05236

[70] Feras Saad, Brian Patton, Matthew Douglas Hoffman, Rif A. Saurous, and Vikash Mansinghka. 2023. Sequential Monte Carlo Learning for Time Series Structure Discovery. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 29473–29489. https://proceedings.mlr.press/v202/saad23a.html

[71] Matteo Salvador, Fanwei Kong, Mathias Peirlinck, David Parker, Henry Chubb, Anne Dubin, and Alison Marsden. 2023. Digital twinning of cardiac electrophysiology for congenital heart disease. *Biorxiv* (2023), 2023–11.

[72] John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (2016), e55.

[73] Amelie Schäfer, Pavanjit Chaggar, Alain Goriely, Ellen Kuhl, and Alzheimer's Disease Neuroimaging Initiative. 2022. Correlating tau pathology to brain atrophy using a physics-based Bayesian model. *Engineering with Computers* 38, 5 (2022), 3867–3877.

[74] Adam Scibor, Ohad Kammar, and Zoubin Ghahramani. 2018. Functional programming for modular Bayesian inference. *Proc. ACM Program. Lang.* 2, ICFP, Article 83 (jul 2018), 29 pages. https://doi.org/10.1145/3236778

[75] John Skilling. 2006. Nested sampling for general Bayesian computation. *Bayesian analysis* 1, 4 (2006), 833–859.

[76] Nikola Surjanovic, Miguel Biron-Lattes, Paul Tiede, Saifuddin Syed, Trevor Campbell, and Alexandre Bouchard-Côté. 2023. Pigeons.jl: Distributed Sampling From Intractable Distributions. *CoRR* (2023). arXiv:2308.09769 [stat.CO] http://arxiv.org/abs/2308.09769v1

[77] Robert H Swendsen and Jian-Sheng Wang. 1986. Replica Monte Carlo simulation of spin-glasses. *Physical review letters* 57, 21 (1986), 2607.

[78] Saifuddin Syed, Alexandre Bouchard-Côté, George Deligiannidis, and Arnaud Doucet. 2019. Non-reversible parallel tempering: a scalable highly parallel MCMC scheme. *arXiv preprint arXiv:1905.02939* (2019). https://arxiv.org/abs/1905.02939

[79] Saifuddin Syed, Vittorio Romaniello, Trevor Campbell, and Alexandre Bouchard-Côté. 2021. Parallel Tempering on Optimized Paths. *CoRR* (2021). arXiv:2102.07720 [stat.CO] http://arxiv.org/abs/2102.07720v2

[80] Sean Talts, Michael Betancourt, Daniel Simpson, Aki Vehtari, and Andrew Gelman. 2018. Validating Bayesian Inference Algorithms With Simulation-Based Calibration. *CoRR* (2018). arXiv:1804.06788 [stat.ME] http://arxiv.org/abs/1804.06788v2

[81] Zhixin Cyrillus Tan, Anja Lux, Markus Biburger, Prabha Varghese, Stephen Lees, Falk Nimmerjahn, and Aaron S Meyer. 2023. Mixed IgG Fc immune complexes exhibit blended binding profiles and refine FcR affinity estimates. *Cell reports* 42, 7 (2023).

[82] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).

[83] Yee Whye Teh, Bryn Elesedy, Bobby He, Michael Hutchinson, Sheheryar Zaidi, Avishkar Bhoopchand, Ulrich Paquet, Nenad Tomasev, Jonathan Read, and Peter J. Diggle. 2022. Efficient Bayesian Inference of Instantaneous Reproduction Numbers At Fine Spatial Scales, With an Application To Mapping and Nowcasting the Covid-19 Epidemic in British Local Authorities. *Journal of the Royal Statistical Society Series A: Statistics in Society* 185, Supplement_1 (2022), S65–S85. https://doi.org/10.1111/rssa.12971

[84] Andrew Thomas. 2006. The BUGS language. *R News* 6 (2006), 17–21. Issue 1. https://journal.r-project.org/articles/RN-2006-005/.

[85] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and implementation of probabilistic programming language anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional programming Languages*.

1–12.

[86] Alan N Tump, Dominik Deffner, Timothy J Pleskac, Pawel Romanczuk, and Ralf HJ M. Kurvers. 2024. A cognitive computational approach to social and collective decision-making. *Perspectives on Psychological Science* 19, 2 (2024), 538–551.

[87] Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner. 2021. Rank-Normalization, Folding, and Localization: an Improved R^ for Assessing Convergence of Mcmc (with Discussion). *Bayesian Analysis* 16, 2 (2021), nil. https://doi.org/10.1214/20-ba1221

[88] Kai Xu, Hong Ge, Will Tebbutt, Mohamed Tarek, Martin Trapp, and Zoubin Ghahramani. 2020. AdvancedHMC. jl: A robust, modular and efficient implementation of advanced HMC algorithms. In *Symposium on Advances in Approximate Bayesian Inference*. PMLR, 1–10.

[89] Lu Zhang, Bob Carpenter, Andrew Gelman, and Aki Vehtari. 2021. Pathfinder: Parallel Quasi-Newton Variational Inference. *CoRR* (2021). arXiv:2108.03782 [stat.ML] http://arxiv.org/abs/2108.03782v2

A EXAMPLES

This section contains some additional code blocks related to the examples mentioned in the main text.

A.1 MS-GARCH

In the two code blocks below, note the additional `Type{ArrayType}` argument with default value `Array{Float64}` passed to the model. When such an argument is specified, `Turing.jl` uses this internally to accommodate potentially different types of data needed in certain situations, e.g. when using `ForwardDiff.jl`'s dual numbers to compute gradients, while still ensuring that the resulting model is type stable. This is not strictly necessary to include in the model definition, but when explicit allocations are being performed, as in the models below, then this can lead to significant performance improvements vs. using abstractly typed containers, e.g. `Matrix{Real}`.

```
@model function ms_garch(
    K,                                  # Number of states.
    T,                                  # Length of the time series.
    ::Type{ArrayType}=Array{Float64},   # Used internally by Turing.jl.
) where {ArrayType}
    □ ~ filldist(Dirichlet(ones(K)), K)  # Transition matrix.
    □ ~ ordered(filldist(□□(0, 0.5), K)) # Initial volatility.
    □ ~ filldist(□□(0, 1, 1), K)         # Volatility persistence.
    □ ~ arraydist(□□.(1, 1, 1 .- □))     # Volatility persistence.

    □ = ArrayType{2}(undef, T, K)        # Volatilities.
    z = Vector{Int}(undef, T)            # Latent state.
    y = ArrayType{1}(undef, T)           # Emission.

    for k in 1:K                         # Initial volatilities.
        □[1, k] = □[k] / (1 - □[k] - □[k] + 1e-6)
    end

    □□ = fill(1/K, K)                    # Initial state probabilities.
    z[1] ~ Categorical(□□)               # Initial state.
    y[1] ~ Normal(0, □[1, z[1]])         # Initial emission.

    for t = 2:T
        z[t] ~ Categorical(□[:, z[t-1]]) # Transition.
        for k = 1:K                      # Update volatilities with GARCH(1, 1) dynamics.
            □[t, k] = □(□[k] + □[k] * y[t-1]^2 + □[k] * □[t-1, k]^2)
        end
        y[t] ~ Normal(0, □[t, z[t]])     # Emission.
    end

    return (; □, z, □, y)
```

Listing 22. Generative process for the MS-GARCH model from Section 3.1 in `Turing.jl`, including discrete parameters.

```julia
@model function ms_garch_marginalized(
    K,                                          # Number of states.
    T,                                          # Length of the time series.
    y,                                          # Time series observations.
    ::Type{ArrayType}=Array{Float64}            # Used internally by Turing.
) where {ArrayType}
    □ ~ filldist(Dirichlet(ones(K)), K)         # Transition matrix.
    □ ~ ordered(filldist(□□(0, 0.5), K))        # Initial volatility.
    □ ~ filldist(□□(0, 1, 1), K)                # Volatility persistence.
    □ ~ arraydist(□□.(1, 1, 1 .- □))            # Volatility persistence.

    □ = ArrayType{2}(undef, T, K)               # Volatilities.
    for k in 1:K                                # Initial volatilities.
        □[1, k] = □[k] / (1 - □[k] - □[k] + 1e-6)
    end
    for t = 2:T
        for k = 1:K                             # GARCH(1, 1) dynamics for volatilities.
            □[t, k] = □(□[k] + □[k] * y[t-1]^2 + □[k] * □[t-1, k]^2)
        end
    end

    □□ = fill(1/K, K)                           # Initial state probabilities
    trans = permutedims(□)                      # Transition matrix.
    emissions = eachrow(Normal.(0, □))          # Emission distributions.
    hmm = TimeHMM(□□, trans, emissions)         # Instantiate the custom HMM type.
    Turing.@addlogprob! logdensityof(hmm, y, 1:T) # Add the marginal log-likelihood.

    return (; □, □, y, y_pred=last(rand(hmm, 1:T)))
end
```

Listing 23. Generative process for the MS-GARCH model from Section 3.1 in `Turing.jl` with discrete parameters marginalized out using `HiddenMarkovModels.jl`.

B INTERNALS EXTENDED

In this section, we dig a bit deeper into the internals of `Turing.jl` and its surrounding ecosystem. This is a section meant for the very curious reader who wants to understand how `Turing.jl` works under the hood at the time of writing. Given that these are internals and not always publicly exposed APIs, we warn the reader that the details in this section might be subject to change in future versions of `Turing.jl`. However, it should provide a good starting point for understanding the general structure of `Turing.jl` and how it is implemented.

B.1 `DynamicPPL.jl`: The DSL

B.1.1 The `@model` macro. To get a better understanding of the `@model` macro, let us consider the following toy-model:

```julia
@model function demo(args...; kwargs...)
    x ~ Normal()
    return "Hello, world!"
end
```

Listing 24. A toy model defined using the `@model` macro.

Here we have a model taking some arbitrary arguments `args...` and keyword arguments `kwargs...`, contains one random variable `x` which is distributed according to `Normal()`, and returns the string ``` ``Hello, world!'' ```. Inspecting the code generated by the `@model` macro can be done using Julia's `@macroexpand`, which results in the following:[9]

```julia
# Evaluator.
function demo(
    __model__::Model,
    __varinfo__::AbstractVarInfo,
    __context__::AbstractContext,
    args...;
    kwargs...,
)
    # BEGIN `x ~ Normal()`
    dist = Normal()
    vn = DynamicPPL.VarName{:x}()
    # Determine if `x ~ Normal()` is an assume statement.
    isassumption = if DynamicPPL.contextual_isassumption(__context__, vn)
        if !DynamicPPL.inargnames(vn, __model__) || DynamicPPL.inmissings(vn, __model__)
            true
        else
            x === missing
        end
    else
        false
    end
    # Perform different actions based on whether `x ~ Normal()` is:
    # 1. A fixed statement.
    # 2. An assume statement.
    # 3. An observe statement.
    if DynamicPPL.contextual_isfixed(__context__, vn)
        # If `x` is fixed => ignore `isassumption` and assign the variable the fixed value.
        x = DynamicPPL.getfixed_nested(__context__, vn)
    elseif isassumption
        # If `x ~ Normal()` is an assume statement => execute the tilde-assume pipeline.
        (value, __varinfo__) = DynamicPPL.tilde_assume!!(
            __context__,
            DynamicPPL.unwrap_right_vn(DynamicPPL.check_tilde_rhs(dist), vn)...,
            __varinfo__,
        )
        x = value
```

[9]This is a cleaned up version of the output with supporting annotations, as the actual output is quite verbose and contains a lot code which is not relevant for this discussion.

```
37          value
38      else
39          # If `x ~ Normal()` is an observe statement => execute the tilde-observe pipeline.
40          if !DynamicPPL.inargnames(vn, __model__)
41              # If `x` is not an argument AND `isassumption` is false => in the `__context__`.
42              x = DynamicPPL.getconditioned_nested(__context__, vn)
43          end
44          (value, __varinfo__) = DynamicPPL.tilde_observe!!(
45              __context__,
46              DynamicPPL.check_tilde_rhs(dist),
47              x,
48              vn,
49              __varinfo__,
50          )
51          value
52      end
53      # END `x ~ Normal()`
54
55      # BEGIN `return "Hello, world!`
56      retval = "Hello, world!"
57      # Include `__varinfo__` to allow immutable implementations of `__varinfo__`.
58      return (retval, __varinfo__)
59      # END `return "Hello, world!`
60  end
61
62  # Constructor.
63  $(Expr(:meta, :doc))
64  function demo(args; kwargs...)
65      return Model(demo, NamedTuple{(:args,)}((args,)); kwargs...)
66  end
```

Listing 25. Annotated expansion of the `@macro` call in Listing 15

In Listing 15 we see that there are indeed two methods generated from the single user-defined function `demo`, as promised.

As mentioned in Section 4, the `constructor` on L63-66 is fairly straight-forward: it takes in the same arguments as the user-provided function, and returns a `DynamicPPL.Model` object, which is subsequently used in downstream processing.

The `evaluator` is quite a bit more complicated. Inspecting L1-7 in Listing 25 we see that, other than the user-provided `args...` and `kwargs...`, the `evaluator` takes in three "internal" arguments: a `DynamicPPL.Model`, a `DynamicPPL.AbstractVarInfo`, and a `DynamicPPL.AbstractContext`. The `__model__::Model` is the model object which is the return-value of the constructor, whose purpose we will describe in more detail later. The `__varinfo__::AbstractVarInfo` is what is sometimes referred to as a *trace* structure in the literature [48], and is used to keep track of the random variables in the model and their realizations, further elaborated on in Section 4.2.4. Finally, the `__context__::AbstractContext` defines the *context* of the evaluation, which enables customization of the evaluation process, e.g. specifying whether we are sampling from the model or simply evaluating the log-probability of a given realization.

In the body of the `evaluator`, the first line of the original model was has been expanded into L9-51 in Listing 25. On lines L10-11 in Listing 25 we see that the left-hand side of `x ~ Normal()` has been parsed into this `VarName{:x}` instance and the right-hand side has been captured in a variable `dist`. `VarName` is described further in a Section 4.2.3.

The next part in L13-52 is determining whether the variable `x` should be considered either of the following: a) a *fixed* variable, i.e. a variable which is not to be sampled, nor to be included in the log-probability calculation, b) a *random* variable which is to be sampled (assume statement), or c) a *conditioned* variable which is not to be sampled, but should be included in the log-probability calculation (observe statement). In short, the category the variable falls within is determined according to the process outlined in Figure 9. The `contextual_isfixed` and `contextual_isassumption` checks if the `__context__` specifies that the variable should be fixed or assumed, respectively, while the `inargnames && x === missing` check is used to determine if the variable is provided as an argument to the model, and if so, whether it evaluates to `missing`, indicating that the user wants it to be sampled (assume), or if it is an observed value (observe). It's important to note that in most typical use-cases, these checks will be completely compiled away, thus having no impact on runtime performance.

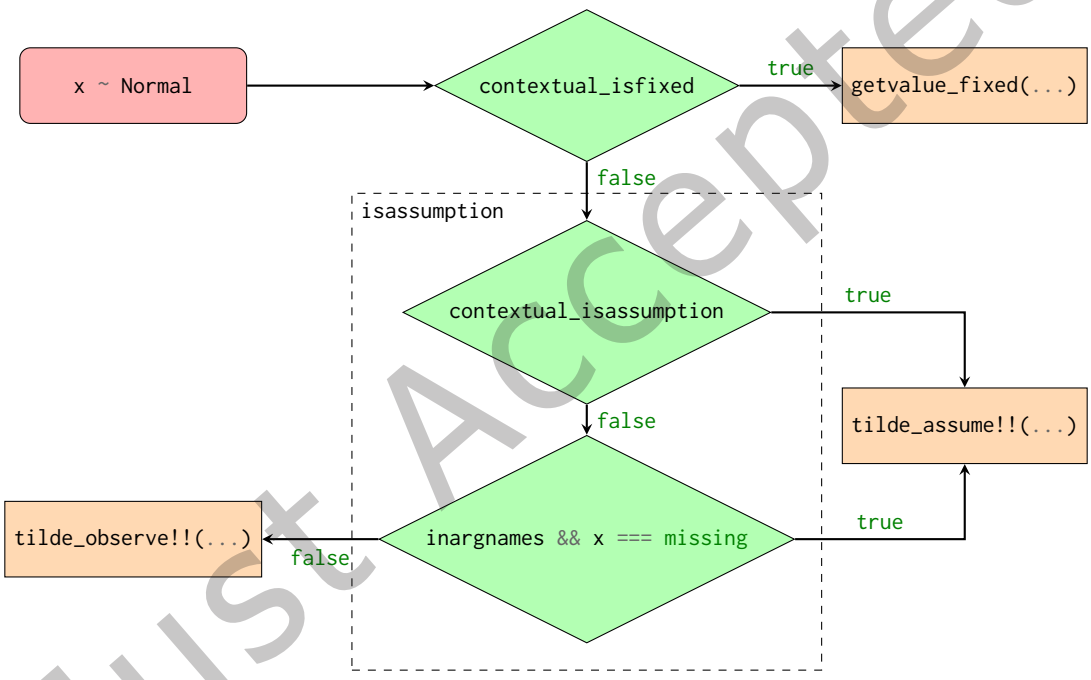

Fig. 9. Decision diagram for determining whether a variable should be considered fixed, assumed, or observed.

Note that in both the case of an `assume` and `observe` statement, the internal variable `__varinfo__` is re-assigned with the (potentially) new trace structure returned by the `tilde_assume!!` and `tilde_observe!!` functions, which are responsible for updating the trace structure `__varinfo__` with incoming information about the variable `x`. This re-assignment makes it possible to support both mutable and immutable trace structures; in the scenario of mutable trace structures, the `__varinfo__` variable is updated in-place, while in the case of immutable trace structures, a new trace structure is returned and re-assigned to `__varinfo__`.

Finally, on L58 we see that the **`return`** value of the `evaluator` has been expanded into **`return`** `(retval, __varinfo__)`. That is, the `evaluator` returns the original expression provided in the model

definition *and* the trace structure `__varinfo__`. This is another necessary step in order to support immutable trace structures. Note that in a "standard" call to the `Model`, e.g. `model()`, the trace structure is dropped before passed back to the caller; it is only explicitly captured in internal evaluations of the model. This way, the user is not exposed to the internal trace structure, but it is still available for internal use.

B.1.2 The tilde-pipeline. First we have the `tilde_assume!!` function which is called when `left` is considered a random variable:

```
function tilde_assume!!(context, right, left, varinfo)
    value, logp, varinfo = tilde_assume(context, right, left, varinfo)
    return value, acclogp_assume!!(context, varinfo, logp)
end
```

Listing 26. `tilde_assume!!`, the entry point for assume statements in `DynamicPPL.jl`.

We see that this immediately delegates to `tilde_assume` which is the function that can be overloaded by contexts to change the behavior of the tilde pipeline and is expected to return the following values:

- `value`: the realization of the random variable `left`.
- `logp`: the log probability of the realization.
- `varinfo`: the, possibly altered, `AbstractVarInfo` object (see Section 4.2.4).

The `value` and `logp` are then passed to `acclogp_assume!!` which is responsible for accumulating the log probability of the realization of the random variable for an assume statement. This `tilde_assume` usually consists of several nested calls to `tilde_assume`, eventually reaching the base case method `assume` for either just log-probability evaluation or for sampling and log-probability evaluation. In effect, ignoring the fact that any context in the context stack can `return` at any point before reaching `assume`, we end up with a call stack with a single root `tilde_assume!!` and only two possible leaf nodes `assume` without sampling or `assume` with sampling, as illustrated in Figure 10.

Similarly, `tilde_observe!!` is the entry point for the observe pipeline, which takes the form shown in Listing 27, where `tilde_observe` is a recursive function that eventually reaches the base case method `observe`, in a similar fashion to `tilde_assume`.

```
function tilde_observe!!(context, right, left, varinfo)
    logp, varinfo = tilde_observe(context, right, left, varinfoe)
    return left, acclogp_observe!!(context, varinfo, logp)
end
```

Listing 27. `tilde_observe!!`, the entry point for observe statements in `DynamicPPL.jl`.

Even though most inference methods absolutely do not need to touch these internals, e.g. `HMC` only really requires a map $x \mapsto \left(\log p(x), \nabla \log p(x)\right)$, it does enable full control over the inference process when needed. For example, particle MCMC methods implemented in `Turing.jl` makes uses continuation passing style to implement the resampling step in particle methods [31]. This is done by overloading the `observe` methods to yield back to the transition step after each observation is processed, allowing communication between the particles, e.g. to resample particles based on the current state of the chain.

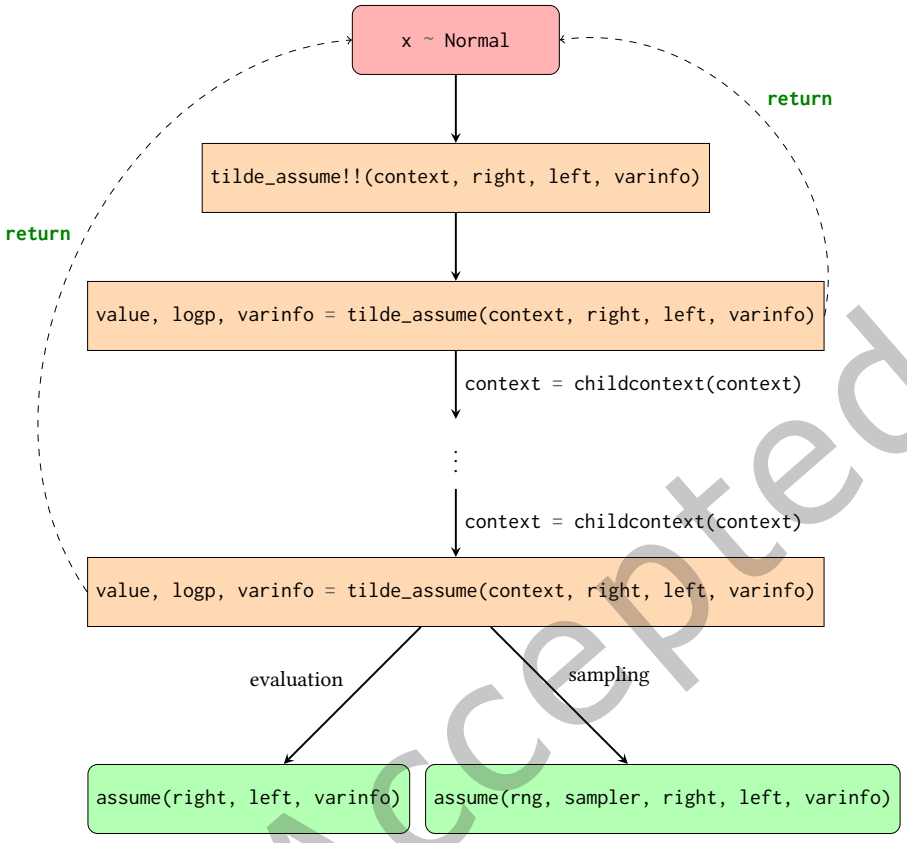

Fig. 10. The call stack resulting from a `x ~ Normal()` when `x` is treated as a random variable. As we can see, the call stack is a tree with a single root `tilde_assume!!` and two possible leaf nodes `assume`, one without sampling and one with sampling. The former is useful for model evaluations which assume variables are already present in the trace structure, while the latter is used for sampling new variables. The intermediate nodes `tilde_assume` allow the contexts present in the context stack to alter the behavior of the tilde pipeline.

B.1.3 Contexts. Inspired by contextual dispatch mechanisms in `Cassette.jl` [67] in addition to the effect handler approach of `Pyro` and `NumPyro`, `DynamicPPL.jl` makes extensive use of different *contexts* to handle different aspects of the modeling and inference process. In `DynamicPPL.jl`, most methods involved in the evaluation of a `Model` takes in an argument with type `AbstractContext` which can be dispatched on to change the behavior of the method. Moreover, a context can itself contain other contexts, producing a stack-like structure of contexts which affect different stages of the call stack. For example, if a context `MyContext <: AbstractContext` wants to alter the tilde pipeline, it would do so as follows:

```julia
# Path hit for `left ~ right` when `left` is considered a random variable.
function tilde_assume(context::MyContext, left, right, varinfo)
    # Do something before passing the arguments to the next context.
    ...
    # Let the next context, indicated as a child of the current `context`, do its thing.
    value, varinfo, logp = tilde_assume(childcontext(context), left, right, varinfo)
    # Do something after the child-context has completed.
    ...
    # Pass the results on to the next context.
    return value, varinfo, logp
end
```

Listing 28. An example of how a context can alter the assume pipeline.

Similarly one can override observe statements by overloading `tilde_observe`. For example, a likelihood weighting context could overload the `tilde_observe` method to multiply the log-probability of the observation by the weight of the context:

```julia
# Path hit for `left ~ right` when `left` is considered an observed variable.
function tilde_observe(context::LikelihoodWeightingContext, left, right, varinfo)
    value, varinfo, logp = tilde_observe(childcontext(context), left, right, varinfo)
    # Weight the `logp` by the `weight` of the context.
    return value, varinfo, logp * context.weight
end
```

Listing 29. An example of how a context can alter the observe pipeline.

Such a likelihood-weighting context could then be used to implement inference methods which rely on annealing the likelihood of a model, e.g. [26, 77, 78]. The context mechanism is used to implement a wide range of features in `DynamicPPL.jl`, such as prefixing of the variable names in the case of submodels to avoid name clashes (as seen in Section 2.4), `condition`, `fix`, etc., in addition to providing a way for more advanced users, e.g. (UG 2) or (UG 3), to implement their own custom model behaviors or inference algorithms.

This mechanism is used to implement a wide range of features in `DynamicPPL.jl`, such as prefixing the variable names in the case of models within models to avoid name clashes (as seen in Section 2.4), `condition`, `fix`, etc., in addition to providing a way for more advanced users, e.g. (UG 2) or (UG 3), to implement their own custom model behaviors or inference algorithms.

B.2 Implementing a sampler

As discussed in the main text, there are two main ways one can implement a sampler to be used with `Turing.jl`: a) target the `LogDensityProblems.jl` interface, and b) overload the behavior of the `~` statements. In both of these cases, we still implement the `AbstractMCMC.jl` `step` interface briefly mentioned in Section 4.3 and Listing 20.

As a very simple example, we will implement two slightly different versions of an importance sampling (IS) method, and target the very simple model:

```julia
@model function demo()
    x ~ Normal()
    y ~ Normal(x, 1)
end
model = demo() | (y = 1,)
```

Listing 30. A very simple model with one random variable x and one observation y.

We've chosen this trivial model to minimize clutter in output and avoid having to deal with minor complexities such as working in constrained vs. unconstrained space and converting into a nice `Chains` object. Such things are easily dealt with in `DynamicPPL.jl` but we avoid them here for the sake of readability.

To implement an IS method on this model, we need to implement the following two steps:

(1) Sample a new state from the proposal q, i.e. $x\ q$.
(2) Compute the (log) weight $\log p(x, y)/q(x)$ for the new state, where $p(x, y) = p(y \mid x)p(x)$ denotes the joint distribution of the target model.

B.2.1 By targeting `LogDensityProblems.jl`. `LogDensityProblems.jl` [61] is a Julia package without any direct connection to `Turing.jl`; the package was developed independently of `Turing.jl` by different developers. All the package does is to specify the following "interface":

- `logdensity(model, x::AbstractVector{<:Real})`: computes the log-density represented by `model` at `x`.
- `dimension(model)`: returns the expected length of the input `x`.
- `capabilities(model)`: specifies whether we have access to gradient information or higher-order differential information.

This benefit of such an interface is that an inference researcher (UG 3) can easily implement their new inference method for this interface, and then subsequently have their method applicable to all the Julia packages that implement this interface. Of course, this interface does not provide much information about the target model, but for many inference methods used in practice this is sufficient, e.g. NUTS[38]. `DynamicPPL.jl`, and thus `Turing.jl`, is one such package that implements this interface exactly to allow users of `Turing.jl` to use inference methods, even if the implementation is only specified for `LogDensityProblems.jl`.

The following snippet demonstrates how to implement IS with a specified proposal by using `AbstractMCMC.jl` targeting a model following the `LogDensityProblems.jl` interface.

```julia
using DynamicPPL, AbstractMCMC, Distributions
using LogDensityProblems: logdensity, dimension

# Define the sampler type.
struct IS{D<:Distribution} <: AbstractMCMC.AbstractSampler
    proposal::D
end

# Convenient constructor: just use a Gaussian as proposal.
IS(model, scale::Real) = IS(MvNormal(zeros(dimension(model)), scale^2 * I))

# Implement the `step` interface from AbstractMCMC.jl.
function AbstractMCMC.step(
    # The random-seed, used to ensure reproducibility.
```

```
    rng::Random.AbstractRNG,
    # Wrapper to indicate support for LogDensityProblems.jl
    logdensity_wrapper::AbstractMCMC.LogDensityModel,
    # Dispatch on our `IS` sampler.
    sampler::IS,
    # No state needed since it's refreshed every `step`.
    state::Nothing=nothing
)
    # Extract the logdensity object.
    model = logdensity_wrapper.logdensity
    # Sample from the proposal given by the sampler.
    x = rand(rng, sampler.proposal)
    # Compute the IS weight.
    weight = logdensity(model, x) - logpdf(sampler.proposal, x)

    # First return-value is the sample, and second is the sampler state.
    # This sampler is independent of the previous iteration, so we just
    # return `nothing` as the state.
    return (params = x, weight = weight), nothing
end

# To sample the Turing.jl `model`, we first reify `model` into an object
# compatible with LogDensityProblems.jl.
f = DynamicPPL.LogDensityFunction(model)
# Then construct the sampler.
sampler = IS(f, 5)
# And call sample.
samples = sample(f, sampler, 2; progress=false)
# Example output:
# 2-element Vector{@NamedTuple{params::Vector{Float64}, weight::Float64}}:
#  (params = [1.1854694394840994], weight = -0.0012622173982026474)
#  (params = [1.738311876787476], weight = -1.0324823613438858)
```

Listing 31. Example implementation of IS with a static proposal by targeting the `LogDensityProblems.jl` interface.

B.2.2 By overloading `~`. The other way to implement a sampler for a `Turing.jl` `Model` is to directly overload the `~` statements; or more specifically, overload the `assume` and `observe` instructions.

But doing this just to implement a standard IS method is a bit over-elaborate given that we just saw how to do this in a straight-forward manner in the previous section. Instead, in this section, we will implement a specific instance of IS called *likelihood weighting* [48]. Likelihood weighting is an IS approach which uses the prior distribution as a proposal. When using the prior as a proposal, we see that the IS weight simplifies as follows

$$\frac{p(x,y)}{p(x)} = \frac{p(y \mid x)p(x)}{p(x)} = p(y \mid x)$$

That is, we only need to keep track of the *likelihood*, not the prior distribution, thus avoiding those pesky `logpdf` computations in the prior. In this case, we require more information about the target model than what the `LogDensityProblems.jl` interface offers (that interface makes no guarantees about the availability of a prior distribution, nor how to interact with it), and so implementing this sampler by overloading `assume` and `observe` is a bit more sensible.

The following snippet shows such an implementation.

```julia
using Random: AbstractRNG
using DynamicPPL: DynamicPPL, VarName, AbstractVarInfo, SamplingContext, evaluate!!

# Definition of the sampler type.
struct LikelihoodWeighting <: AbstractMCMC.AbstractSampler end

# Here we overload the `assume` instruction when sampling for
# our `LikelihoodWeighting` sampler.
function DynamicPPL.assume(
    rng::AbstractRNG,
    sampler::LikelihoodWeighting,
    dist::Distribution,
    vn::VarName,
    varinfo::AbstractVarInfo
)
    # Sample from the prior.
    val = rand(rng, dist)
    # Update the value in `varinfo`.
    varinfo[vn] = val
    # Don't compute the `logpdf` because we'll divide this out.
    return val, 0, varinfo
end

# We could also overload `DynamicPPL.observe`, but the default implementation
# includes the log-likelihood, so in this case we don't need to.

# Implement the `step` interface from AbstractMCMC.jl.
function AbstractMCMC.step(rng::AbstractRNG, model::Model, sampler::LikelihoodWeighting)
    # Construct the initial trace structure for `model`.
    varinfo = DynamicPPL.typed_varinfo(rng, model)
    # Then we return a convenient representation as the first value, and the sampler
    # state as the second. Here we'll simply use the trace structure itself as the state.
    # We use `values_as` to extract the realizations in a nice format.
    return (params=values_as(varinfo, OrderedDict), logweight=getlogp(varinfo)), varinfo
end

function AbstractMCMC.step(
    rng::AbstractRNG, model::Model, sampler::LikelihoodWeighting, varinfo::AbstractVarInfo
)
    # We evaluate the `model` using the `SmaplingContext`, providing our `sampler`.
    # `evaluate!!` is evaluates the `Model`, returning both the `return` specified
    # in `@model` but also the internal `__varinfo__`. We capture the latter.
    varinfo = last(evaluate!!(model, varinfo, SamplingContext(rng, sampler)))
    # Then we return a convenient representation as the first value, and the sampler
    # state as the second. We use `values_as` to extract the realizations in a nice format.
    return (params=values_as(varinfo, OrderedDict), logweight=getlogp(varinfo)), varinfo
end

# Instantiate the sampler.
sampler = LikelihoodWeighting()
# Call `sample`.
sample(model, sampler, 2; progress=false)
# Example output:
```

```
# 2-element Vector{@NamedTuple{...}}:
#  (params = OrderedDict(x => 0.36422878339212106), logweight = -2.106310889668529)
#  (params = OrderedDict(x => -0.4465922419673132), logweight = -1.9652530904646817)
```

Listing 32. Example implementation of likelihood weighting by overloading `assume` and `observe`.

L29 in Listing 32 also demonstrates the approach of evaluating the model once to obtain a type stable trace structure which is reused for subsequent computations, as discussed in Section 4.2.4. For the target in Listing 30 this is completely fine to do, this model is static.

Received 2 May 2024; revised 22 November 2024; accepted 1 December 2024