5th Symposium on Advances in Approximate Bayesian Inference, 2023 1–21

Automatically Marginalized MCMC in Probabilistic Programming

Jinlin Lai JINLINLAI@CS.UMASS.EDU
Javier Burroni JBURRONI@CS.UMASS.EDU
Hui Guan HUIGUAN@CS.UMASS.EDU
Daniel Sheldon SHELDON@CS.UMASS.EDU
University of Massachusetts Amherst

Abstract

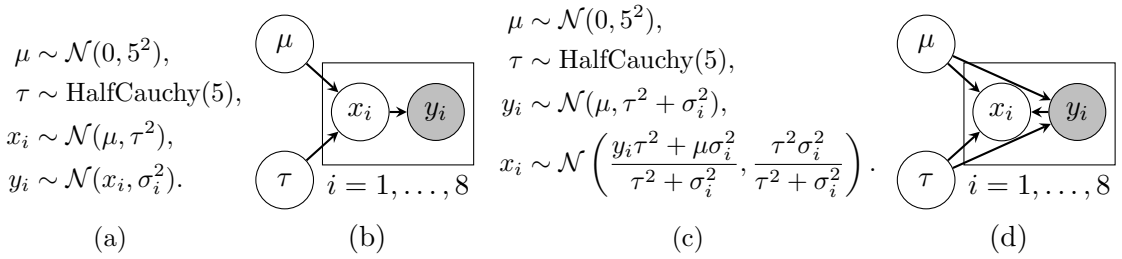
Hamiltonian Monte Carlo (HMC) is a powerful algorithm to sample latent variables from Bayesian models. The advent of probabilistic programming languages (PPLs) frees users from writing inference algorithms and lets users focus on modeling. However, many models are difficult for HMC to solve directly, which often require tricks like model reparameterization. We propose to use automatic marginalization as part of the sampling process using HMC in a graphical model extracted from a PPL, which substantially improves sampling from real-world hierarchical models.

1. Introduction

Probabilistic programming languages (PPLs) promise to automate the inference in Bayesian reasoning for users. We focus on a setting of PPLs that has had large impact in practice, where a model is compiled to a differentiable log-density function for inference by a variant of Hamiltonian Monte Carlo (HMC) (Duane et al., 1987; Neal, 1996). Our methods would likely benefit other Markov chain Monte Carlo (MCMC) inference approaches as well. We focus in particular on *generative* PPLs that correspond to a (directed) graphical model, which means that random variables are defined according to a fixed sequence of conditional distributions. This includes most applied statistical models written in generative PPLs such as Pyro (Bingham et al., 2018), NumPyro (Phan et al., 2019), PyMC (Patil et al., 2010), Edward (Tran et al., 2017) and TensorFlow Probability (Piponi et al., 2020). It does not directly include Stan programs, which do not always specify a sampling procedure, though most can be converted to do so (Baudart et al., 2021).

Despite their promise, the barrier between users and inference in PPLs is often blurred. There may be different ways to write a model, with inference performance depending critically on the specific choice, such that users again need specialized knowledge. One issue is: it is often possible to reformulate a generative model so that some latent variables are generated *after* all observed variables, which allows them to be dropped during MCMC and then reconstructed afterward. We refer to this as *marginalization*, because the variables are marginalized while running MCMC. By reducing the number of variables for MCMC, marginalization can lead to substantial performance gains. However, it places a significant burden on the user to reformulate the model.

We develop a method to automatically marginalize variables in a user-specified probabilistic program for inference with HMC. Our work builds on prior research on automatic



(a) (b) (c) (d)

Figure 1: Formulas and graphical models of the original and reformulated eight schools models. (a), (b): original model; (c), (d): reformulated model.

marginalization (see Section 4) and shares technical underpinnings with work to automatically Rao-Blackwellize particle filters for evaluation-based PPLs (Murray et al., 2018; Atkinson et al., 2022). The difference is we focus on graphical models and HMC, which leads to different technical considerations. Although most HMC-based PPLs compile directly to a log-density, we use the program-tracing features of JAX (Bradbury et al., 2018) to extract a graphical-model representation of programs written in NumPyro. In the graphical model, we identify conjugacy relationships and manipulate the graphical model to marginalize some variables. HMC is run on the reduced model, and the marginalized variables are recovered by direct sampling conditional on the samples from HMC. Importantly, the interface between the user and the PPL does not change.

2. Motivating examples

We first present an example model where marginalization can significantly benefit HMC-based inference. The eight schools model (Gelman et al., 1995) is an important demonstration model for PPLs (Gorinova, 2022) and reparameterization (Papaspiliopoulos et al., 2007). It is a hierarchical model to study the effect of coaching on SAT performance in eight schools. Mathematically, the model is in Figure 1 (a), where $i \in \{1, \ldots, 8\}$ and $(\sigma_{1:8}, y_{1:8})$ are given as data. We want to reason about all latent variables, μ, τ and $x_{1:8}$. A PPL will compile the model code to a log joint density $\log p(\mu, \tau, x_{1:8}, y_{1:8})$ and then run HMC over the latent variables μ, τ and $x_{1:8}$.[1] However, there is another reformulated model with the same joint density shown in Figure 1 (c). Both models are shown as graphical models in Figure 1 (b) and (d): they have different causal interpretations but identical joint distributions and are therefore the same for performing inference. Importantly, since only $y_{1:8}$ are observed, it is possible to marginalize $x_{1:8}$ to obtain the reduced model $p(\mu, \tau, y_{1:8}) = p(\mu)p(\tau)\prod_{i=1}^{8} p(y_i \mid \mu, \tau)$. We can sample μ and τ by running HMC on the reduced model then sample $x_{1:8}$ directly from $p(x_{1:8} \mid \mu, \tau, y_{1:8})$ given μ and τ. With this strategy, HMC samples 2 variables instead of 10, which significantly speeds up inference.

The principle that allows us to transform the model is conjugacy. In a Bayesian model $p(x, y) = p(x)p(y \mid x)$ the prior $p(x)$ is conjugate to the likelihood $p(y \mid x)$ if the posterior $p(x \mid y)$ is in the same parametric family as $p(x)$ for all y. For our working definition, we assume the parametric families of the prior and likelihood have a tractable density function and sampling procedure and that there is an analytical formula for the parameters of the posterior in terms of y. Given these assumptions, it is also possible to sample from the

1. In practice, latent variables are transformed to have real support (Kucukelbir et al., 2017).

marginal $p(y)$ and compute its density efficiently.[2] Conjugacy is formally a property of distribution families, but we will also say "x is conjugate to y" when the meaning is clear from context. In the eight schools model, x_i is conjugate to y_i given μ and τ, which leads to analytical expressions for the distributions $p(x_i \mid \mu, \tau, y_i)$ and $p(y_i \mid \mu, \tau)$ in the reformulated model and ensures they have tractable densities and samplers. We wish to automate this procedure so users only write the original model and our framework reformulates it.

3. Automatically marginalized MCMC

Given a program written by a user, our method will construct a graphical model and then manipulate it into a reformulated model for which MCMC samples fewer variables. The key operation will be *reversing* certain edges (based on conjugacy) to create unobserved leaf nodes that can be marginalized. For example, in the eight schools model of Figure 1, the edge from x_i to y_i is reversed, after which x_i is a leaf. In this section, we develop the algorithm assuming a suitable graphical model representation.

3.1. Graphical model representation

Assume there are M random variables $x_1, x_2, \ldots, x_M$. For a set of indices A, we write $\mathbf{x}_A = (x_i)_{i \in A}$. A graphical model G is defined by specifying a distribution family for each node together with a mapping from parents to parameters. Specifically, for node i, let D_i represent its distribution family from a finite set of options (e.g., "Normal", "Beta", etc.), let $\mathrm{pa}(i) \subseteq \{1, \ldots, M\}$ be its parents, and let $f_i \colon \mathcal{X}_{\mathrm{pa}(i)} \to \Theta_i$ be a mapping such that x_i has distribution $D_i(\theta_i)$ with parameters $\theta_i = f(\mathbf{x}_{\mathrm{pa}(i)})$. With this representation, given concrete values of all variables, the log density can be computed easily as $\sum_{i=1}^{M} \log p_i\big(x_i \mid f_i(\mathbf{x}_{\mathrm{pa}(i)})\big)$, assuming nodes are ordered topologically. Generating a joint sample is similar: iterate through nodes and sample $x_i \sim h_i\big(\,\cdot \mid f_i(\mathbf{x}_{\mathrm{pa}(i)})\big)$.

3.2. Marginalizing unobserved leaf nodes

As a first useful transformation of the graphical model, we consider how to improve HMC if there is an unobserved leaf node. Without loss of generality, assume the leaf is numbered M. Then we can factor the joint distribution as $p(\mathbf{x}_{1:M}) = p(\mathbf{x}_{1:M-1})p(x_M \mid \mathbf{x}_{1:M-1})$ and run HMC on the marginalized model $p(\mathbf{x}_{1:M-1})$, then sample x_M directly from $p(x_M \mid \mathbf{x}_{1:M-1})$ by executing $h_M(\,\cdot \mid f_M(\mathbf{x}_{\mathrm{pa}(M)}))$. Importantly, the marginal $p(\mathbf{x}_{1:M-1}) = \prod_{i=1}^{M-1} p_i(x_i \mid f_i(\mathbf{x}_{\mathrm{pa}(i)}))$ is simply the original graphical model with the leaf node deleted, so it is tractable. More generally, the argument is easily extended by repeatedly stripping leaves to marginalize all variables with no path to an observed variable for HMC, then to reconstruct those variables by *ancestral sampling* from the graphical model (Koller and Friedman, 2009).

3.3. Marginalizing non-leaf nodes by edge reversals

Generative models such as the eight schools model do not have unobserved leaf nodes in their original forms. Instead, our goal will be to transform the model by a sequence of *edge reversals* to create unobserved leaf nodes. Each edge reversal will preserve the joint

2. To sample, draw $x \sim p(x), y \sim p(y \mid x)$ and ignore x; for the density, use $p(y) = \frac{p(x_0)p(y \mid x_0)}{p(x_0 \mid y)}$ for any x_0.

distribution of the graphical model, so it is the same for performing inference. However, it will not preserve the causal semantics of the data-generating process (which is not required for inference), so it is reasonable for the transformed model to have unobserved leaf nodes.

Reversing a single edge. The process of reversing a single parent-child edge $v \to c$ is illustrated in Figure 2. There must be no other path from v to c; otherwise reversing the edge would create a cycle. In the example, there is no other path because v has only one child. Let us define the "local distribution" of x_v and x_c as the product of the conditional distributions of those two variables given their parents, which is $p(x_v \mid \mathbf{x}_{\mathrm{pa}(v)})p(x_c \mid x_v, \mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}})$. If these distributions satisfy the appropriate conjugacy relationship, we can derive replacement factors $p(x_c \mid \mathbf{x}_U)p(x_v \mid x_c, \mathbf{x}_U)$, where $U = \mathrm{pa}(v) \cup \mathrm{pa}(c) \setminus \{v\}$, to "reverse" the $v \to c$ edge while preserving the local distribution.

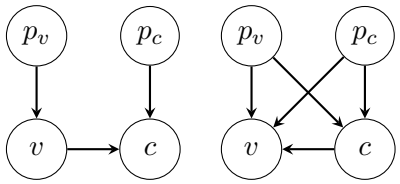


Figure 2: Reversing edge $v \to c$.

It is easy to show that edge reversal yields a graphical model with the same joint distribution as the original. To understand the utility of this operation, observe in Figure 2 that node v becomes a leaf and can be marginalized after reversing $v \to c$. In principle, any edge can be reversed, but it is only tractable when one can derive the replacement factors. We can do so if the distributions are *locally conjugate*:

Definition 1 (Local conjugacy):
Let G be a graphical model where node v is a parent of c. We say the distribution of x_v is *locally conjugate* to the distribution of x_c if $\hat{p}(x_v) := p(x_v \mid \mathbf{x}_{\mathrm{pa}(v)})$ is conjugate to $\hat{p}(x_c \mid x_v) := p(x_c \mid x_v, \mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}})$ for all values of $\mathbf{x}_{\mathrm{pa}(v)}$ and $\mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}}$.

The details of edge reversal using our graphical model representation and specific conjugate pairs of distribution families will be discussed in Appendix B.

Creating a leaf by reversing all outgoing edges of a node. We next consider how, if possible, to convert an arbitrary node v to a leaf by reversing all of its outgoing edges. Suppose v has H children $c_1, \ldots, c_H$. If c is minimal among $c_1, \ldots, c_H$ in a topological ordering of G, then there can be no other $v \to c$ path, so it is safe to reverse $v \to c$. Further, after reversing the edge, v will move in the topological ordering to appear after c but before the other children, without changing the relative ordering of the other children. Then another child will be minimal in the topological ordering. Therefore, if it is possible to convert v to a leaf, we should reverse the edges from v to each of its children following their topological ordering. The formal version of this reasoning is proved in Appendix C.

Marginalizing many non-leaf nodes. The previous part describes how to modify a graphical model, while preserving the joint distribution, to convert one non-leaf node to a leaf so it can be marginalized. We now wish to use this operation to marginalize as many nodes as possible. The MARGINALIZE function in Algorithm 1 presents our heuristic for doing so: it simply applies the operation of marginalizing one node to attempt to marginalize every node v in *reverse* topological order. This is convenient because it automatically strips all nodes with no path to an observed variable at the same time. If v can be marginalized, the reversal operations are executed and v is removed from G and pushed onto a stack S that determines the recovery order. The implementations of CONJUGATE and REVERSE are discussed in Appendix B.

Algorithm 1 Marginalize and recover unobserved nodes

function MARGINALIZE (G)
  Initialize stack S and sort nodes so they are numbered in topological order
  for each unobserved node v in descending order **do**
    if CONJUGATE(G, v, c) for all children c **then**
      for each child c in ascending order **do**
        G = REVERSE(G, v, c) // Marginalize v by reversing edges
      Remove v from G and add v to top of S
  return G, S

4. Related work

Conjugacy and marginalization have long been important topics in probabilistic programming. Related works include Hakaru (Narayanan et al., 2016), PSI (Gehr et al., 2016, 2020), Autoconj (Hoffman et al., 2018), delayed sampling, and semi-symbolic inference (Atkinson et al., 2022). Please refer to Appendix D for a detailed discussion of these works.

5. Experiments

We use NumPyro's no-U-turn sampler (NUTS) (Hoffman and Gelman, 2014) in all experiments, denoted HMC hereafter. Our approach is "HMC with marginalization" (HMC-M). For all experiments, we use 10,000 warm up samples to tune the sampler, 100,000 samples for evaluation, and evaluate performance via effective sample size (ESS) and time (inclusive of JAX compilation time).

5.1. Hierarchical partial pooling models

A hierarchical partial pooling (HPP) model (Gelman et al., 1995) has the form $p(\theta, z_{1:n}, y_{1:n}) = p(\theta)\prod_{i=1}^{n} p(z_i \mid \theta)p(y_i \mid \theta, z_i, x_i)$, where (x_i, y_i) are observed covariate and response values for the ith data point, z_i is a local latent variable, and θ is a global latent variable to model shared dependence. One application of HPPs is repeated binary trials, where we observe the number of successes y_i out of K_i trials for each unit i, and assume a partially shared structure for the success probabilities, such as (Carpenter et al., 2017):

$$m \sim \text{Uniform}(0, 1),\ \kappa \sim \text{Pareto}(1, 1.5),\ \theta_i \sim \text{Beta}(m\kappa, (1-m)\kappa),\ y_i \sim \text{Binomial}(K_i, \theta_i).$$

Applications include the rat tumors dataset (Tarone, 1982), the baseball hits 1970 dataset (Efron and Morris, 1975) and the baseball hit 1996 AL dataset (Carpenter et al., 2017). This model is difficult for HMC due to a funnel relationship between κ and θ_i (Carpenter et al., 2017). Suggested remedies are to model κ with an exponential distribution (Patil et al., 2010) or rewrite the model to one where reparameterization is applicable (Carpenter et al., 2017).

We observe that, since θ_i (Beta) is locally conjugate to y_i (Bernoulli), marginalization is a better strategy. In the marginalized model, y_i is a beta-binomial random variable, HMC samples only m and κ, and each θ_i is sampled afterward from $p(\theta_i \mid m, \kappa, y_i)$, a beta distribution. The funnel problem is eliminated and the HMC dimension is reduced from $n + 2$ to 2. Our methods achieve this automatically.

Table 1: Evaluation metrics for HMC and HMC-M on the repeated binary trials model. Mean and std over 5 independent runs are reported.

Dataset	Algorithm	Min ESS	Time (s)	Min ESS/s
Baseball hits 1970 ($n = 18$)	HMC	1384.1 (1156.7)	**94.5** (5.7)	14.8 (12.7)
	HMC-M	**39001.8** (20030.4)	110.5 (89.2)	**592.3** (304.2)
Rat tumors ($n = 71$)	HMC	24632.3 (1494.5)	654.8 (43.9)	37.7 (2.1)
	HMC-M	**77644.5** (9570.8)	**72.4** (0.3)	**1072.7** (134.0)
Baseball hits 1996 AL ($n = 308$)	HMC	9592.3 (260.1)	2746.1 (107.6)	3.5 (0.2)
	HMC-M	**61109.0** (3344.9)	**130.9** (1.4)	**467.0** (29.5)

$$\mu_i \sim \mathcal{N}(0,1),\ a_j \sim \mathcal{N}(100\mu_{\text{gp}[j]},1),$$
$$b_i \sim \mathcal{N}(0,100^2),\ \log\sigma_i \sim \mathcal{N}(0,1),$$
$$y_k \sim \mathcal{N}(a_{p_k} + t_k b_{g_k}, \sigma^2_{g_k}).$$

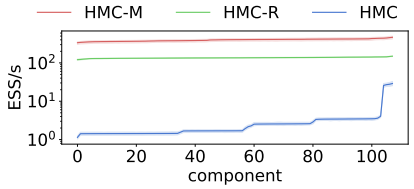


Figure 3: Expressions and component-wise ESS/s (ordered) for the electric company model. HMC-M is compared against HMC and HMC with reparameterization (HMC-R).

Table 1 shows the results. Sampling κ is known to be difficult in this model, but HMC-M achieves an ESS with similar magnitude to the number of samples. The HMC problem dimension is also reduced, which leads to faster running time. These factors combined lead to more than 100x ESS/s improvement on the baseball hit 1996 AL data set.

5.2. Hierarchical linear regression

Similar to partial pooling, hierarchy can be introduced in linear regression models. The electric company model (Gelman and Hill, 2006) studies the effect of an educational TV program on children's reading abilities. There are $C = 192$ classes in $G = 4$ grades divided into $P = 96$ treatment-control pairs. Class k is represented by (g_k, p_k, t_k, y_k) where g_k is the grade, p_k is the index of pair, $t_k \in \{0,1\}$ is the treatment variable and y_k is the average score. The classes in pair j belong to grade $\text{gp}[j]$. The full model is in Figure 3 left, where $i \in \{1,\dots,G\}$, $j \in \{1,\dots,P\}$ and $k \in \{1,\dots,C\}$. Observe that μ_i, a_j, b_i and y_k are all normally distributed with affine dependencies. Therefore, it is possible to marginalize μ_i, a_j and b_i from the HMC process.

We observed that marginalization of μ_i led to very high JAX compilation times even though the computation graph for the log-density was not much larger than the one before marginalization (14606 primitive operations vs. 9186). We attribute this to a current JAX limitation. See Appendix F for experimental evidence. As a workaround, we manually prevented μ_i from being marginalized. Figure 3 right shows the results. In this model, HMC performs poorly, but reparameterizing the a_j variables is very helpful: HMC-R achieves excellent ESS (comparable to the number of samples). However, HMC-R does not reduce the dimension and solves a $3G + P = 108$ dimension problem, while automatic marginalization reduces the problem dimension to 8 and results in an additional 4x speed up.

References

Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and Michael Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. *Proc. ACM Program. Lang.*, 6(OOPSLA), 2022.

Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 898–912, 2020.

Guillaume Baudart, Javier Burroni, Martin Hirzel, Louis Mandel, and Avraham Shinnar. Compiling Stan to generative probabilistic languages and extension to deep probabilistic programming. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 497–510, 2021.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

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. JAX: composable transformations of Python+NumPy programs. 2018.

Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

Marco Cusumano-Towner, Alexander K Lew, and Vikash K Mansinghka. Automating involutive MCMC using probabilistic and differentiable programming. *arXiv preprint arXiv:2007.09871*, 2020.

Arnaud Doucet, Nando De Freitast, and Stuart Russent. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*, pages 176–183. Citeseer, 2000.

Simon Duane, Anthony D Kennedy, Brian J Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2):216–222, 1987.

Bradley Efron and Carl Morris. Data analysis using Stein's estimator and its generalizations. *Journal of the American Statistical Association*, 70(350):311–319, 1975.

Timon Gehr, Sasa Misailovic, and Martin Vechev. PSI: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.

Timon Gehr, Samuel Steffen, and Martin Vechev. λPSI: Exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2020. doi: 10.1145/3385412.3386006.

Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006.

Andrew Gelman, John B Carlin, Hal S Stern, and Donald B Rubin. *Bayesian data analysis*. Chapman and Hall/CRC, 1995.

Maria Gorinova, Dave Moore, and Matthew Hoffman. Automatic reparameterisation of probabilistic programs. In *International Conference on Machine Learning*, pages 3648–3657. PMLR, 2020.

Maria I Gorinova. *Program Analysis of Probabilistic Programs*. PhD thesis, University of Edinburgh, 2022.

Maria I Gorinova, Andrew D Gordon, Charles Sutton, and Matthijs Vákár. Conditional independence by typing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 44(1):1–54, 2021.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Matthew Hoffman, Pavel Sountsov, Joshua V Dillon, Ian Langmore, Dustin Tran, and Srinivas Vasudevan. Neutra-lizing bad geometry in Hamiltonian Monte Carlo using neural transport. *arXiv preprint arXiv:1903.03704*, 2019.

Matthew D. Hoffman and Andrew Gelman. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

Matthew D Hoffman, Matthew J Johnson, and Dustin Tran. Autoconj: recognizing and exploiting conjugacy without a domain-specific language. *Advances in Neural Information Processing Systems*, 31, 2018.

Kurt Hornik, Friedrich Leisch, Achim Zeileis, and M Plummer. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. In *Proceedings of DSC*, volume 2, 2003.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *J. Mach. Learn. Res.*, 18:14:1–14:45, 2017.

Daniel Lundén. Delayed sampling in the probabilistic programming language Anglican, 2017.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS-a Bayesian modeling framework: concepts, structure, and extensibility. *Statistics and computing*, 10 (4):325–337, 2000.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric involutive Markov chain Monte Carlo. In *International Conference on Machine Learning*, pages 14802–14859. PMLR, 2022.

Lawrence M Murray and Thomas B Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic Rao-Blackwellization of probabilistic programs. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, volume 84 of *Proceedings of Machine Learning Research*, pages 1037–1046. PMLR, 2018.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 62–79. Springer, 2016.

Radford M Neal. *Bayesian learning for neural networks*, volume 118 of *Lecture Notes in Statistics*. Springer-Verlag, 1996.

Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry Vetrov. Involutive MCMC: a unifying framework. In *International Conference on Machine Learning*, pages 7273–7282. PMLR, 2020.

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P Chen. Functional tensors for probabilistic programming. *arXiv preprint arXiv:1910.10775*, 2019a.

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Neeraj Pradhan, Justin Chiu, Alexander Rush, and Noah Goodman. Tensor variable elimination for plated factor graphs. In *International Conference on Machine Learning*, pages 4871–4880. PMLR, 2019b.

George Papamakarios, Eric T. Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing flows for probabilistic modeling and inference. *J. Mach. Learn. Res.*, 22:57:1–57:64, 2021.

Omiros Papaspiliopoulos, Gareth O Roberts, and Martin Sköld. A general framework for the parametrization of hierarchical models. *Statistical Science*, pages 59–73, 2007.

Matthew D Parno and Youssef M Marzouk. Transport map accelerated Markov chain Monte Carlo. *SIAM/ASA Journal on Uncertainty Quantification*, 6(2):645–682, 2018.

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

Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

Dan Piponi, Dave Moore, and Joshua V Dillon. Joint distributions for Tensorflow probability. *arXiv preprint arXiv:2001.11819*, 2020.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

Robert E Tarone. The use of historical control information in testing for a trend in proportions. *Biometrics*, pages 215–220, 1982.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

Appendix A. Difficulty of manual reformulation

Hierarchical linear regression. The eight schools model is very simple, but already requires user effort to reformulate. To emphasize the complexity of reformulating larger models, here we present a simplified version of the electric company model (Gelman and Hill, 2006). The full model appears in Section 5.2. The original model is

$$\log \sigma \sim \mathcal{N}(0,1),\ \mu_a \sim \mathcal{N}(0,1),\ a \sim \mathcal{N}(100\mu_a, 1),\ b_{1:2} \sim \mathcal{N}(0, 100^2),\ y_i \sim \mathcal{N}(a + b_i t_i, \sigma^2).$$

And the observed variables are y_1 and y_2. One can guess that the model can be reformulated because, conditioned on σ, all variables are normal with means that are affine functions of other variables. However, the calculations are complex: some variables of the reformulated model will be

$$a \sim \mathcal{N}\left(\frac{100\mu_a\sigma^2 + y_1 + y_2 - b_1 t_1 - b_2 t_2}{2+\sigma^2}, \frac{\sigma^2}{2+\sigma^2}\right),$$
$$b_1 \sim \mathcal{N}\left(\frac{100^2 t_1 (y_1 - 100\mu_a)}{1 + 100^2 t_1^2 + \sigma^2}, \frac{100^2 + 100^2\sigma^2}{1 + 100^2 t_1^2 + \sigma^2}\right),$$
$$b_2 \sim \mathcal{N}\left(\frac{100^2 t_2 (y_2 + \sigma^2 y_2 - y_1 - 100\sigma^2\mu_a + b_1 t_1)}{100^2 t_2^2 + 100^2\sigma^2 t_2^2 + 2\sigma^2 + \sigma^4}, \frac{2 * 100^2\sigma^2 + 100^2\sigma^4}{100^2 t_2^2 + 100^2\sigma^2 t_2^2 + 2\sigma^2 + \sigma^4}\right).$$

In this version, HMC is run to sample μ_a and σ (distributions not shown) conditioned on y_1 and y_2. Then a, b_1, and b_2 are reconstructed conditioned on μ_a, σ, y_1, y_2 by sampling from the shown distributions. By reducing the number of variables from 5 to 2, HMC inference can be accelerated. However, it is extremely cumbersome for the user to derive the new model, which no longer corresponds to the originally conceived data generating process.

Appendix B. Details of automatically marginalized MCMC

B.1. Computation graph representation

Our operations to transform the graphical model will require examining and manipulating the functions $f_i(\mathbf{x}_{\mathrm{pa}(i)})$ mapping parents to distribution parameters. For example, in the simplified electric company model, we need to detect from the symbolic expression $y_2 \sim \mathcal{N}(a + b_2 t_2, \sigma^2)$ that the mean parameter is an affine function of b_2, which is required to reverse the edge $b_2 \to y_2$. Similarly, we must manipulate symbolic expressions to obtain ones like those in the reformulated model. For this purpose we assume functions are represented as computation graphs.

Consider an arbitrary function $f(x_{i_1}, x_{i_2}, \ldots, x_{i_k})$ for $i_1, \ldots, i_k \in \{1, \ldots, m\}$. We assume the computation graph of f is specified as a sequence of N_f *primitive operations* that each write one value (Griewank and Walther, 2008), which is similar to the *JAX expression (Jaxpr)* representation we can obtain from JAX. Specifically, the sequence of values $w_1, w_2, w_3, \ldots, w_{k+N_f}$ are computed as follows: (1) the first k values are the inputs to the function, i.e., $w_j = x_{i_j}$ for $j = 1$ to k, and (2) each subsequent value is computed from the preceding ones as $w_j = \phi_j(\mathbf{w}_{\mathrm{pred}(j)})$, where ϕ_j is a primitive operation (e.g., "ADD", "MUL", "SQUARE") on values $\mathbf{w}_{\mathrm{pred}(j)}$ and $\mathrm{pred}(j) \subseteq \{1, \ldots, j-1\}$ is the set of predecessors of j. The predecessor relationship defines a DAG for the variables in a computation graph.

Table 2: Patterns of conjugacy. If conditions of a row are satisfied, then the distribution of x_a is locally conjugate to the distribution of x_b.

Distribution of x_a	Distribution of x_b	Condition 1	Condition 2
$\mathcal{N}(\mu_a, \sigma_a^2)$	$\mathcal{N}(\mu_b, \sigma_b^2)$	AFFINE(μ_b, x_a)	not DEPENDENT(σ_b, x_a)
$\Gamma(\alpha_a, \beta_a)$	$\Gamma(\alpha_b, \beta_b)$	LINEAR(β_b, x_a)	not DEPENDENT(α_b, x_a)
$\Gamma(\alpha_a, \beta_a)$	Exponential(λ_b)	LINEAR(λ_b, x_a)	-
Beta(α_a, β_a)	Binomial(n_b, p_b)	$p_b = x_a$	not DEPENDENT(n_b, x_a)
Beta(α_a, β_a)	Bernoulli(λ_b)	$\lambda_b = x_a$	-

We will also need to algorithmically manipulate computation graphs. In the text, we will denote manipulations symbolically as follows. Suppose $f(x_A)$ and $g(x_B)$ are two functions represented by computation graphs with potentially overlapping sets of input variables. We use expressions such as $f * g$ or $f + g$ to mean the new computation graph representing this symbolic expression. For example the computation graph for $f + g$ has input variables $x_{A \cup B}$ and consists of the graphs for f and g together with one additional node (primitive operation) for the final addition.

B.2. Conjugacy detection

Detecting when x_a is locally conjugate to x_b uses the patterns listed in Table 2, where (1) AFFINE(u, v) means that u can be written as $u = pv + q$ for expressions p and q that do not contain v, (2) DEPENDENT(u, v) means that there exists a path from v to u in the computation graph, and (3) LINEAR(u, v) means that u can be written as $u = pv$, for an expression p that does not contain v. For example, the pattern in the first matches the case when x_a and x_b both have normal distributions, in which case we can extract expressions (computation graphs) for the parameters μ_b and σ_b^2 as the two outputs of the expression $f_b(\mathbf{x}_{\text{pa}(b)})$. The pattern further implies that if AFFINE(μ_b, x_a) is true and DEPENDENT(σ_b^2, x_a) is false, then x_a is locally conjugate to x_b. The functions AFFINE, LINEAR and DEPENDENT require examining computation graphs.

Conjugacy detection requires looking into the computation graph of functions. In the following part, we introduce how conjugacy detection is performed on a computation graph. During the tracing of a program, the procedure of computation is compiled into intermediate representations consisting of basic operations. For example, the function

```
def f(x, y):
    m = x - y
    n = x + y
    p = m ** 2
    q = n ** 2
    return p + q
```

could be represented as

Algorithm 2 Determining dependency of a variable on an input. $\mathbf{x}_{1:M}$ is the set of all random variables.

function DEPENDENT (w_j, x)
  if $w_j = x$ **then**
    return True
  if $w_j \in \mathbf{x}_{1:M}$ **then**
    return False
  for $p \in \text{pred}(j)$ **do**
    if DEPENDENT(w_p, x) **then**
      return True
  return False

```
INPUTS: a, b
c = SUB a b
d = SQUARE c
e = ADD a b
f = SQUARE e
g = ADD d f
OUTPUTS: g
```

By looking at the intermediate representations, it is possible to reason about the relationship between outputs and inputs. We have reduced conjugacy detection to affinity, linearity and dependency detections. We first introduce the details of dependency detection with the above definition in Section B.1.

Given a function with a computation graph, we may want to determine whether a variable w_j depends on an input x_a. We define the result to be DEPENDENT(w_j, x_a), which could be obtained recursively through the equations shown in Algorithm 2. Note that in this paper, the inputs of functions are always random variables in $\mathbf{x}_{1:M}$. For DEPENDENT(w_j, x_a), if w_j is a random variable in $\mathbf{x}_{1:M}$, then it must be an input of the function. Then DEPENDENT would return whether w_j is x_a. If w_j is the result of a basic operator, it would enumerate the inputs of that operator. If any of those inputs are dependent on x_a, then the result is true. The recursive algorithm could be of exponential complexity in some special cases. We store intermediate results in a dictionary and refer to it before recursion to avoid redundant computation. Then the complexity is linear with respect to the number of variables involved.

Affinity and linearity can be included in the same framework. In addition to determining whether w_j is affine to x_a, we further return whether the slope and intercept are non-zero. If affinity is detected with zero intercept, the relationship is then linear. We define AFFINE_ALL(w_j, x_a) to be a tuple of three bool variables - whether w_j is affine on x_a, whether the slope is non-zero and whether the intercept is non-zero. Then LINEAR and AFFINE could be obtained from AFFINE_ALL(w_j, x_a). Our algorithm of affinity detection is adapted from Atkinson et al. (2022) with slight modification to setting that has no concrete values. The pseudocodes are in Algorithm 3. The result of AFFINE_ALL(w_j, x_a) could be obtained by enumeration of cases of ϕ_j and induction in the structure of the

Algorithm 3 Determining affinity and linearity of a variable on an input. $\mathbf{x}_{1:M}$ is the set of all random variables.

function AFFINE_ALL (w_j, x)
  if $w_j = x$ **then**
    return True, True, False
  if $w_j \in \mathbf{x}_{1:M}$ **then**
    return True, False, True
  if $\phi_j \in \{\text{ADD}, \text{SUB}\}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $r_1, s_2, t_1 = \text{AFFINE\_ALL}(w_{p_1}, x)$
    $r_2, s_2, t_2 = \text{AFFINE\_ALL}(w_{p_2}, x)$
    return r_1 **and** r_2, s_1 **or** s_2, t_1 **or** t_2
  if $\phi_j = \text{MUL}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $r_1, s_2, t_1 = \text{AFFINE\_ALL}(w_{p_1}, x)$
    $r_2, s_2, t_2 = \text{AFFINE\_ALL}(w_{p_2}, x)$
    if not s_1 **then**
      return r_1 **and** r_2, t_1 **and** s_2, t_1 **and** t_2
    if not s_2 **then**
      return r_1 **and** r_2, s_1 **and** t_2, t_1 **and** t_2
    return False, False, False
  if $\phi_j = \text{DIV}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $r_1, s_2, t_1 = \text{AFFINE\_ALL}(w_{p_1}, x)$
    $r_2, s_2, t_2 = \text{AFFINE\_ALL}(w_{p_2}, x)$
    if not s_2 **then**
      return r_1 **and** r_2, s_1, t_1
    return False, False, False
  for $p \in \text{pred}(j)$ **do**
    $r, s, t = \text{AFFINE\_ALL}(w_p, x)$
    if not r **or** s **then**
      return False, False, False
  return True, False, True

function AFFINE (w_j, x)
  $r, s, t = \text{AFFINE\_ALL}(w_j, x)$
  return r

function LINEAR (w_j, x)
  $r, s, t = \text{AFFINE\_ALL}(w_j, x)$
  return r **and not** t

computation graph. For example, if we know $w_j = \text{ADD}(w_{p_1}, w_{p_2})$, and $r_1, s_1, t_1 = \text{AFFINE\_ALL}(w_{p_1}, x_a)$ and $r_2, s_2, t_2 = \text{AFFINE\_ALL}(w_{p_2}, x_a)$, then w_j is affine to x_a if both of w_{p_1} and w_{p_2} are affine to x_1, which means (r_1 **and** r_2). The slope is non-zero if any of the slope of p_1 or p_2 is non-zero, so the second return value is (s_1 **or** s_2). The same applies to whether the intercept is non-zero, which is (t_1 **or** t_2).

B.3. Edge reversal details

If conjugacy is detected, Algorithm 1 will call the REVERSE operation to reverse an edge. Algorithm 4 shows the REVERSE algorithm for each case. We emphasize that operations like $+$, $-$ and $*$ are symbolic operations on computation graphs. For the normal-normal case, the algorithm implements well known Gaussian marginalization and conditioning formulas. Line 5 extracts the symbolic expressions for the parameters of the normal distributions. Line 6 extracts expressions p and q such that $\mu_c = px_v + q$; conjugacy detection has already determined that such expressions exist. Lines 7–12 compute symbolic expressions for parameters of the marginal $p(x_c|\cdots)$ and conditional $p(x_v|x_c, \cdots)$ and write them to f_c and f_v. Finally, Lines 36–37 update the DAG to reflect the new dependencies.

In Algorithm 4 a function AFFINE_COEFF is defined to get the coefficients when affinity is detected. The pseudocode of it is in Algorithm 5, which is similar to AFFINE. We again emphasize that the computations in Algorithm 5 are fully symbolic. Each variable corresponds to a sequence of operations which could be regarded as a computation graph. One issue is we need to define whether some variables are zero (lines 17,19,25). So operations of zeros should be specially dealt with. For example, if we find a $0 + 0$, instead of declaring an operation that adds two zeros, we should instead use the result 0.

B.4. Implementation

We have assembled the pieces for automatically marginalized HMC. The full pipeline is to: (1) extract a graphical model G from the user's program, (2) call the MARGINALIZE function to get a marginalized model G' and recovery stack S, (3) run HMC on G', (4) for each HMC sample $\mathbf{x}$, call RECOVER$(S, \mathbf{x})$ to sample the marginalized variables.

Our implementation uses JAX (Bradbury et al., 2018) and NumPyro (Bingham et al., 2018; Phan et al., 2019) to extract a graphical model G. We use JAX tracing utilities to convert the NumPyro program to a JAX expression (Jaxpr), i.e., computation graph, for the entire sampling procedure. The NumPyro program must use a thin wrapper around NumPyro's sample statement to register the model's random variables in the Jaxpr. We extract the distribution families from the NumPyro trace stack and obtain the parameter functions f_i by parsing the Jaxpr to extract the partial computation mapping from parent random variables to distribution parameters. As stated earlier, our approach is limited to programs that map to a graphical model, which means they sample from a fixed sequence of conditional distributions. This closely matches those programs for which NumPyro can currently perform inference, because the JIT-compilation step of NumPyro inference requires construction of a static computation graph. NumPyro's experimental control flow primitives ("scan" and "cond") are not supported, and it may be difficult to do so. Our current implementation is limited to Jaxprs with scalar operations and elementwise array operations, though this restriction is not fundamental. We expect our approach is compatible with other PPLs that use computation graphs, with similar restrictions on programs.

Appendix C. Formal definition of edge reversal

The operation of edge reversal can be formally defined as follows:

Algorithm 4 Reversing an edge

function REVERSE ($G = (D_i, \text{pa}(i), f_i)_{i=1}^{M}$,$v$,$c$)
  // $\text{pa}(v), \text{pa}(c), f_v, f_c, D_c$ updated in place
  // all variables represent symbolic expressions
  if D_c is normal **and** D_v is normal **then**
    Let μ_v and σ_v^2 be the two output expresssions f_v, and μ_c and σ_c^2 be the output expressions of f_c.
    $p, q = \text{AFFINE\_COEFF}(\mu_c, x_v)$
    $k = \sigma_v^2 p / (p^2\sigma_v^2 + \sigma_c^2)$
    $\mu'_c = p\mu_v + q$
    $\sigma'^2_c = p^2\sigma_v^2 + \sigma_c^2$
    $\mu'_v = \mu_v + k(x_c - \mu'_c)$
    $\sigma'^2_v = (1 - kp)\sigma_v^2$
    $f_c = (\mu'_c, \sigma'^2_c),\ f_v = (\mu'_v, \sigma'^2_v)$
  if D_v is Beta **and** $D_c \in \{\text{Bernoulli}, \text{Binomial}\}$ **then**
    Let α_v and β_v be the two output expressions of f_v
    if $D_c = \text{Bernoulli}$ **then**
      $n_c = 1$
      $p_c = \lambda_c$
    else
      Let n_c and p_c be the two output expressions of f_c
    $\alpha'_v = \alpha_v + x_c$
    $\beta'_v = \beta_v + n_c - x_c$
    $D_c = \text{BetaBinomial}$
    $f_c = (n_c, \alpha_v, \beta_v),\ f_v = (\alpha'_v, \beta'_v)$
  if D_v is Gamma **and** $D_c \in \{\text{Exponential}, \text{Gamma}\}$ **then**
    Let α_v and β_v be the two output expressions of f_v
    if $D_c = \text{Exponential}$ **then**
      $\alpha_c = 1$
      $\beta_c = \lambda_c$
    else
      Let α_c and β_c be the two output expressions of f_c
    $p, q = \text{AFFINE\_COEFF}(\beta_c, x_v)$
    $\alpha'_v = \alpha_v + \alpha_c$
    $\beta'_v = \beta_v + p * x_c$
    $D_c = \text{CompoundGamma}$
    $f_c = (\alpha_c, \alpha_v, \beta_v/p),\ f_v = (\alpha'_v, \beta'_v)$
  $\text{pa}(c) = (\text{pa}(c) \setminus \{v\}) \cup \text{pa}(v)$
  $\text{pa}(v) = \text{pa}(v) \cup \{c\} \cup \text{pa}(c)$
  return G

Definition 1 (Edge reversal):
Assume G is a graphical model where node v is a parent of c and there is no other path from v to c. Reversing the $v \to c$ edge replaces factors $p(x_v \mid \mathbf{x}_{\text{pa}(v)})p(x_c \mid x_v, \mathbf{x}_{\text{pa}(c)\setminus\{v\}})$ by

Algorithm 5 Getting the coefficients of affine relationship between a variable w_j on an input x. $\mathbf{x}_{1:M}$ is the set of all random variables.

function AFFINE_COEFF(w_j, x)
  if $w_j = x$ **then**
    return 1, 0
  if $w_j \in \mathbf{x}_{1:M}$ **then**
    return 0, 1
  if $\phi_j = \text{ADD}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $s_1, t_1 = \text{AFFINE\_COEFF}(w_{p_1}, x)$
    $s_2, t_2 = \text{AFFINE\_COEFF}(w_{p_2}, x)$
    return $s_1 + s_2$, $t_1 + t_2$
  if $\phi_j = \text{SUB}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $s_1, t_1 = \text{AFFINE\_COEFF}(w_{p_1}, x)$
    $s_2, t_2 = \text{AFFINE\_COEFF}(w_{p_2}, x)$
    return $s_1 - s_2$, $t_1 - t_2$
  if $\phi_j = \text{MUL}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $s_1, t_1 = \text{AFFINE\_COEFF}(w_{p_1}, x)$
    $s_2, t_2 = \text{AFFINE\_COEFF}(w_{p_2}, x)$
    if s_1 is zero **then**
      return $t_1 * s_2$, $t_1 * t_2$
    if s_2 is zero **then**
      return $s_1 * t_2$, $t_1 * t_2$
    raise Error
  if $\phi_j = \text{DIV}$ **and** $\text{pred}(j) = \{p_1, p_2\}$ **then**
    $s_1, t_1 = \text{AFFINE\_COEFF}(w_{p_1}, x)$
    $s_2, t_2 = \text{AFFINE\_COEFF}(w_{p_2}, x)$
    if s_2 is zero **then**
      return s_1/t_2, t_1/t_2
    raise Error
  return 0, w_j

$p(x_c \,|\, \mathbf{x}_U)p(x_v \,|\, x_c, \mathbf{x}_U)$ and updates the parent sets as $\text{pa}'(c) = U$, $\text{pa}'(v) = U \cup \{c\}$, where $U = \text{pa}(v) \cup \text{pa}(c) \setminus \{v\}$.

We show that the operation yields a graphical model with the same joint distribution as the original:

Proof It is enough to show that (1) the graphical model after reversal is still valid; (2) the joint distribution does not change, which requires

$$p(x_v \,|\, \mathbf{x}_{\text{pa}(v)})p(x_c \,|\, x_v, \mathbf{x}_{\text{pa}(c)\setminus\{v\}}) = p(x_c \,|\, \mathbf{x}_U)p(x_v \,|\, x_c, \mathbf{x}_U).$$

For (1), we need to show that no cycles could be formed during the process. For any $p_v \in \text{pa}(v)$, an edge $p_v \to c$ is added. Because there does not exist a path from c to p_v (otherwise there will be a loop in the original model), this edge will not cause a loop. For any $p_c \in \text{pa}(c) \setminus \{v\}$, an edge $p_c \to v$ is introduced. This edge will also not cause a loop;

otherwise another path from v to c will be found. Finally, the edge $v \to c$ is replaced with $c \to v$, this edge will also not introduce a loop because there are no other paths from v to c.

Now we show (2). Since there is no other paths from v to c, there is no path from v to any nodes in $\mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}}$. Conditioned on $\mathbf{x}_{\mathrm{pa}(v)}$, by conditional independence, we have that $p(x_v \,|\, \mathbf{x}_{\mathrm{pa}(v)}) = p(x_v \,|\, \mathbf{x}_U)$. Also, all paths from nodes in $\mathrm{pa}(v)$ to c are blocked either by v, or by a parent of c, so conditioned on $\mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}}$, by independence, $p(x_c \,|\, x_v, \mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}}) = p(x_c \,|\, x_v, \mathbf{x}_U)$. By the properties of conjugacy, we have that

$$\begin{aligned} p(x_v \,|\, \mathbf{x}_{\mathrm{pa}(v)}) p(x_c \,|\, x_v, \mathbf{x}_{\mathrm{pa}(c)\setminus\{v\}}) &= p(x_v \,|\, \mathbf{x}_U) p(x_c \,|\, x_v, \mathbf{x}_U) \\ &= p(x_c \,|\, \mathbf{x}_U) p(x_v \,|\, x_c, \mathbf{x}_U). \end{aligned}$$

■

Next we give the following theorem on how to turn a non-leaf node to a leaf.

Theorem 1 *Let G be a graphical model where node v has children $c_1, \ldots, c_H$. If x_v is locally conjugate to each of $x_{c_1}, \ldots, x_{c_H}$, then node v can be turned into a leaf by sorting $c_1, \ldots, c_H$ by any topological ordering and reversing the edges from v to each child following this ordering.*

Proof Without loss of generality, assume $c_1, \ldots, c_H$ are sorted according to topological ordering. We prove by induction. Assume for $k \in \{0, \ldots, H\}$, we have reversed the edges $v \to c_1, \ldots, v \to c_k$, and have the following properties:

(1) The children of v are $c_{k+1}, \ldots, c_H$.

(2) $c_{k+1}, \ldots, c_H$ are ordered topologically;

(3) x_v is a local conjugate prior for each of $x_{c_{k+1}}, \ldots, x_{c_H}$.

We show that the edge $v \to c_{k+1}$ is reversible and the properties still hold for $k+1$ after the reversal. By (1) and (2), c_{k+1} is minimal among the children of v in topological order, so there does not exist a path from v to c_{k+1} other than $v \to c_{k+1}$. By (3), x_v is a local conjugate prior to $x_{c_{k+1}}$. Then we can apply edge reversal to $v \to c_{k+1}$. Now we check all the conditions after the replacement. For property (1), the children of v now become $c_{k+2}, \ldots, c_H$. For property (2), the nodes with edges that changed (either incoming or outgoing) were v, c_{k+1}, and their parents; these nodes all preceded $c_{k+2}, \ldots, c_H$ in topological order prior to the reversal and continue to do so afterward, so the relative ordering of $c_{k+2}, \ldots, c_H$ does not change. For (3), the distribution family of x_v does not change, and the conditional distribution of each of $x_{c_{k+2}}, \ldots x_{c_H}$ does not change. So all conditions of local conjugacy in Table 2 will not change for them, which means the distribution of x_v is still a local conjugate prior for the distributions of each of $x_{c_{k+2}}, \ldots x_{c_H}$.

In summary, the three conditions holds for $k = H$ by induction, which means v can be converted to a leaf following the said procedure. ■

Appendix D. Discussion of related work

In BUGS (Lunn et al., 2000) and JAGS (Hornik et al., 2003), conjugacy was used to improve automatic Gibbs sampling. Hakaru (Narayanan et al., 2016) and PSI (Gehr et al., 2016, 2020) use symbolic integrators to perform marginalization for the purposes of exact inference. We make use of information provided by graphical models to identify certain patterns, which is more efficient in large scale models. Autoconj (Hoffman et al., 2018) proposes a term-graph rewriting system that can be used for marginalizing a log joint density with conjugacy. Our approach is distinct in that we operate on the graphical model and computation graph for the generative process, as opposed to the log-density. Gorinova et al. (2021) propose an information flow type system that could be applied to automatic marginalization of discrete random variables. Following the exploration of more expressive PPLs, streaming models have attracted much attention. Murray et al. (2018) proposed delayed sampling, which uses automatic marginalization to improve inference via the Rao-Blackwellized particle filter (RBPF) (Doucet et al., 2000). Delayed sampling has been developed in Birch (Murray and Schön, 2018), Pyro (Bingham et al., 2018) with funsors (Obermeyer et al., 2019a,b), Anglican (Lundén, 2017) and ProbZelus (Baudart et al., 2020). Atkinson et al. (2022) propose semi-symbolic inference, which further expands the applicability of delayed sampling to models with arbitrary structure. Our work is distinct in that we statically analyze a model prior to performing inference for the purpose of improving MCMC: this makes our approach "fully symbolic" (no concrete values are available) and leads to different algorithmic considerations, though our Algorithm 1 shares technical underpinnings with the hoisting algorithm in Atkinson et al. (2022); see Appendix E for more details.

There are many works that improve HMC inference in PPLs from different perspectives. Stan (Carpenter et al., 2017) has had tremendous impact using HMC inference for PPLs. Because Stan programs specify a log-density and not a sampling procedure, our idea does not directly apply to Stan programs. However, many Stan programs are generative in spirit, and Baudart et al. (2021) characterize a subset of Stan programs on which the methods of this paper can be applied directly. Papaspiliopoulos et al. (2007) propose a general framework for non-centered (re)parameterization in MCMC. Gorinova et al. (2020) automate the procedure of choosing parameterizations of models using variational inference. In Parno and Marzouk (2018) and Hoffman et al. (2019), the parameterizations of all latent variables are learned as normalizing flows (Papamakarios et al., 2021; Rezende and Mohamed, 2015). In models where some variables are marginalizable, our method works better than reparameterization: see Section 5.2 for an example. Mak et al. (2022) use the framework of involutive MCMC (Neklyudov et al., 2020; Cusumano-Towner et al., 2020) to extend the applicability of MCMC to non-parametric models in PPLs.

Appendix E. Relations to the hoisting algorithm

The hoisting algorithm in Atkinson et al. (2022) is an online algorithm that can be used for automatically running Rao-Blackwellized particle filters (RBPF) (Doucet et al., 2000). Our Algorithm 1 is highly related to the hoisting algorithm. Both algorithms can perform conjugacy detection and marginalize all possible random variables. One apparent difference is that Algorithm 1 is written as loops while the hoisting algorithm uses recursions. However,

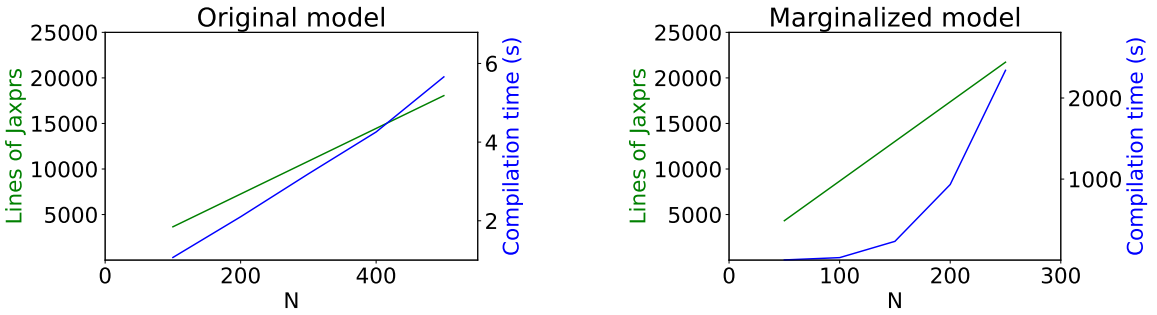


Figure 4: Lines of Jaxprs of the gradient of the log density function and JIT compilation time with respect to N for the simple hierarchical model. With similar lines of Jaxprs, the compilation time can be hundreds of times slower on the marginalized model than on the original model.

the main difference between the two algorithms comes from the application. In RBPF, all non-marginalizable random variables are sampled from the model, allowing the representations to be semi-symbolic, where non-marginalizable random variables are replaced with sampled values during the execution of the hoisting algorithm. In HMC, no random variables are directly sampled, and the same computation graph will be executed many times, so marginalization should be performed before running with fully symbolic representations. In the mean time, Theorem 1 allows us to separate the conjugacy detections and the reversals in two different loops, which reduces the running time in large scale models. This improvement is not possible with the hoisting algorithm as an online algorithm, so some unnecessary reversals are performed. Furthermore, from the perspective of implementation, the parent node is fixed inside the loop of v in Algorithm 1. So in one iteration, all the calls to CONJUGATE and REVERSE are with respect to the same v. It is therefore possible to save the intermediate results of these functions to avoid redundant computation on the computation graph. So the time complexity of Algorithm 1 is $O(M|C|)$, where $|C|$ is the size of the computation graph. With the above considerations, we think that Algorithm 1 is an important contribution in the area.

Appendix F. Slow compilation of JAX

In the experiments, we discover that the compilation time of JAX can be slow for some models. We identify the problem specifically at the structure of marginalized hierarchical models. To demonstrate, we consider the simple model

$$x \sim \mathcal{N}(0,1),\ \log\sigma \sim \mathcal{N}(0,1),\ y_i \sim \mathcal{N}(x,\sigma^2),$$

where $i = 1,\dots,N$ and $y_i = 0$ for all i are provided as pseudo observations. It is possible to marginalize x by reversing edges to each of y_i. However, we found that the JIT compilation time scales super-linear with respect to N for the marginalized model. See Figure 4. Regardless of the performance, the JIT compilation time for the gradient function of the marginalized model can be hundreds larger than that of the original model when N is large

enough, with similar lines of Jaxprs. This is probably because marginalization creates a chain shaped computation graph for all the observations, and it is difficult for JAX to work in this case. We do not regard it as a core limitation of our idea.