3rd Symposium on Advances in Approximate Bayesian Inference, 2020 1–15

Expectation Programming

Tim Reichelt REICHELT@ROBOTS.OX.AC.UK
Adam Goliński ADAMG@ROBOTS.OX.AC.UK
Luke Ong LO@CS.OX.AC.UK
Tom Rainforth RAINFORTH@STATS.OX.AC.UK
University of Oxford

Abstract

Building on ideas from probabilistic programming, we introduce the concept of an *expectation programming framework* (EPF) that automates the calculation of expectations. Analogous to a probabilistic program, an expectation program is comprised of a mix of probabilistic constructs and deterministic calculations, between which a conditional distribution over internal variables and outputs is defined. However, the focus of the inference engine in an EPF is to directly calculate the expectation of the program return values, rather than this conditional distribution. This is made possible by exploiting recent advancements in *target–aware* Bayesian inference, through which we can tailor our inference engines to this expectation estimation, providing the potential for substantial improvements over the standard probabilistic programming pipeline. We realize a particular instantiation of our EPF concept by extending the probabilistic programming language *Turing* with a new `@expectation` macro that uses a series of program transformations to automatically run target–aware inference. We show that this leads to significant empirical gains in estimation performance compared to conventional use of Turing on two example problems.

1. Introduction

Calculating expectations is at the center of many scientific workflows. For example, the decision theoretic foundations of most statistical paradigms, e.g. Bayesian modeling, are rooted in calculating the expectation of a loss function (Robert and Casella, 2004). Moreover, these expectations are often taken with respect to distributions whose densities are only known up to a normalizing constant, requiring some form of inference to be performed.

One such context is probabilistic programming (Gordon et al., 2014; Rainforth, 2017; van de Meent et al., 2018). Here our program is typically specified (often indirectly) through an unnormalized density $\gamma(x)$. The role of system's inference engine is now to approximate the normalized density $\pi(x) = \gamma(x)/Z$, where Z is a normalizing constant equal to the integral of $\gamma(x)$ over all possible values of x. Here if one wishes to calculate some expectation $\mathbb{E}_{\pi(x)}[f(x)]$, the standard pipeline is to construct such an approximation $\hat{\pi}(x)$ independently of $f(x)$, before using this approximation to in turn estimate the expectation.

Sometimes, $f(x)$ is not yet known at the point of inference, such that this pipeline is inevitable. However, in recent work considering a general estimation context (i.e. any arbitrary $\pi(x)$ and $f(x)$), Golinski et al. (2019); Rainforth et al. (2020) showed that if $f(x)$ is known upfront, then we can improve on this pipeline. Namely, they showed that substantial gains can be achieved by using the information in $f(x)$ to construct target-aware Bayesian inference (TABI) estimators directly geared towards the target expectation.

However, current probabilistic programming languages (PPL) (Bingham et al., 2019; Carpenter et al., 2017; Cusumano-Towner et al., 2019; Ge et al., 2018; Salvatier et al., 2016; Tran et al., 2016; Wood et al., 2014), in general, do not allow the explicit incorporation of a function $f(x)$ into their inference calculations. Most languages do not even have a direct consideration of including the definition of such a $f(x)$ into the program definition. Though a handful of papers (Nori et al., 2015; Narayanan et al., 2016; Staton et al., 2016; Zinkov and Shan, 2017) have considered formalizations for the expectation defined by a probabilistic program—typically through considering its return values—none of these do this from the perspective of directly targeting the calculation of this expectation. As such, no existing language current supports generating estimates in a *target–aware* fashion.

To address this, we propose the concept of an expectation programming framework (EPF), defining this as something which directly allows the definition of expectations and then automates their estimation using target-aware inference algorithms. We further introduce a specific implementation of an expectation programming framework, called **EPT** (Expectation Programming in Turing), that is built upon the Turing PPL (Ge et al., 2018). EPT takes as input Turing–style programs and uses program transformations to create a new set of valid Turing programs that are suitable for the construction of TABI estimators, exploiting the fact that the generalised TABI framework of (Rainforth et al., 2020) can repurpose any marginal likelihood estimator into a target-aware inference algorithm.

To realize the potential of EPT, we introduce a new annealed importance sampling (AnIS) (Neal, 2001) inference engine for Turing programs. We then use it to show that our framework can be used to express real expectation problems, and empirically estimate them significantly more accurately than the conventional probabilistic programming pipeline.

2. Background

We are concerned with computing expectation values of the form $\mathbb{E}_{\pi(x)}[f(x)]$ where $f(x)$ is known, but $\pi(x)$ cannot be directly evaluated or sampled from. Namely, $\pi(x) = \gamma(x)/Z$ where $\gamma(x)$ is known unnormalized distribution, but Z is an unknown normalization constant. The standard workflow to estimate $\mathbb{E}_{\pi(x)}[f(x)]$ is to approximate $\pi(x)$ (e.g. with Monte Carlo samples) and then use this to approximate the expectation in turn.

However, this pipeline is suboptimal if we know f ahead of time (Golinski et al., 2019). The Target-Aware Bayesian Inference (TABI) framework (Rainforth et al., 2020) provides a means of creating an estimator for $\mathbb{E}_{\pi(x)}[f(x)]$ that incorporates information from the target function f. TABI breaks down the expectation into three different parts

$$\begin{aligned} &\mathbb{E}_{\pi(x)}[f(x)] = (Z_1^+ - Z_1^-)/Z_2 \\ \text{where}\quad &Z_1^+ = \int \gamma(x) f^+(x)dx, \quad Z_1^- = \int \gamma(x) f^-(x)dx, \quad Z_2 = \int \gamma(x)dx, \\ &f^+(x) = \max(f(x), 0), \quad f^-(x) = -\min(f(x), 0). \end{aligned} \tag{1}$$

We can view the three subcomponents as the normalisation constants of the three densities $\gamma_1^+(x) \propto \gamma(x)f^+(x)$, $\gamma_1^-(x) \propto \gamma(x)f^-(x)$ and $\gamma_2(x) = \gamma(x)$, respectively. Thus we can repurpose any algorithm which provides estimates of the normalisation constant into a

```
@expectation function expct(y)
    x ~ MvNormal(zeros(length(y)), I)         # x ~ N(x; 0, I)
    y ~ MvNormal(x, I)                        # y ~ N(y; x, I)
    return pdf(MvNormal(x, 0.5*I), -y)        # f(x) = N(-y; x, 1/2 I)
end

expct_estimate, diagnostics = estimate_expectation(
  expct(-2*ones(10)/sqrt(10)),
  method=TABI(
    marginal_likelihood_estimator=TuringAlgorithm(AnIS(), num_samples=1000)))
```

Figure 1: An example of estimating an expectation with EPT using a TABI estimator with marginal likelihood estimation of each term performed using 1000 samples of annealed importance sampling (AnIS). `expct(-2*ones(10)/sqrt(10))` yields a 10–dimensional variant of the problem defined in (2). Appendix D shows an equivalent procedure using plain Turing.

target-aware inference algorithm. Namely, we can break down the problem into these three densities, run a marginal likelihood estimator for each (returning $\hat{Z}_1^+$, $\hat{Z}_1^-$, and $\hat{Z}_2$ respectively), and then recombine these to an overall estimator as $\mathbb{E}_{\pi(x)}[f(x)] \approx (\hat{Z}_1^+ - \hat{Z}_1^-)/\hat{Z}_2$.

Throughout the next section we will rely on basic understanding of the Turing PPL (Ge et al., 2018) which we build upon. For the relevant introduction to Turing, see Appendix B.

3. Expectation Programming in Turing

Probabilistic programs can be interpreted as defining expectations over their return values (Gordon et al., 2014). Current probabilistic programming language largely ignore these semantics and focus on approximating conditional distributions. We propose the concept of Expectation Programming Framework (EPF) which exploits this fact to automate the use of target-aware inference algorithms.

We enable users to write *expectation programs* which define an expectation $\mathbb{E}_{\pi(x)}[f(x)]$ by defining an unnormalised density $\gamma(x)=\pi(x)Z$ and a function $f(x)$. Our implementation of the EPF concept, Expectation Programming in Turing (EPT), allows users to specify $\gamma(x)$ as they normally would using Turing `@model` macro, but it interprets the `return` semantics as defining the integrand $f(x)$.

Figure 1 shows how to define and estimate an expectation with EPT. `@expectation` is an EPT macro, which is a drop-in replacement for the Turing `@model` macro, that allows EPT to interpret the `return` semantics without any changes required to the function body with respect to the plain Turing model definition. The expectation `expct` is estimated with the EPT function `estimate_expectation(expct, method)` where `method` specifies the estimation method. EPT implements algorithms which exploit the TABI framework. As per Section 2, TABI requires a choice of a marginal likelihood estimator. In this example we use `TABI` with annealed importance sampling `AnIS`, which is a plain Turing inference algorithm. Importantly, `AnIS` could be substituted with any other Turing inference algorithm that returns a marginal likelihood estimate. In this example, `AnIS()` implies the use of some arbitrary default AnIS parameters regarding the Markov chain Monte Carlo (MCMC) algorithm, and the number and spacing of intermediate potentials used. The final element

```
@expectation function expct(y)
    x ∼ MvNormal(zeros(length(y)), I)
    y ∼ MvNormal(x, I)
    return pdf(MvNormal(x, 0.5*I), -y)
end
```

```
@model function expct(y)
    x ∼ MvNormal(zeros(length(y)), I)
    y ∼ MvNormal(x, I)
    tmp = pdf(MvNormal(x, 0.5*I), -y)
    @addlogprob!(log(max(tmp, 0)))
    return tmp
end
```

Figure 2: The results of one of the three program transformations applied to the EPT `@expectation` program from Figure 1 [left]. Presented is the transformation into a valid Turing `@model` program [right] corresponding to the density $\gamma_1^+(x) \propto \gamma(x)f^+(x)$. The transformed code fragment is highlighted. Appendix E shows the full source code transformation for this model.

of the EPT API is `TuringAlgorithm` which is an object storing the necessary information that allows `TABI` to use a Turing inference method to estimate each of the three terms required by the TABI estimator.

We note that other implementations of the EPF concept can take advantage of other methods that make use of the knowledge of the target function $f(x)$ to improve the process of estimation, in a similar way that different PPLs focus on different families of inference algorithms. For example, we conceive EPF implementations which focus on the use of control variates or other variance reduction techniques.

3.1. Program Transformation

The TABI framework requires us to estimate the normalisation constants of the three densities $\gamma_2(x)$, $\gamma_1^+(x)$ and $\gamma_1^-(x)$. We use Julia's metaprogramming capabilities to transform the user provided `@expectation` model definition into three valid Turing `@model` programs corresponding to each of these densities. Since these are valid Turing models we can *use any inference algorithm implemented in Turing* that provides marginal likelihood estimates. This is a big advantage because it means we do not require custom implementations of inference algorithms that are adapted to our framework.

Firstly, note that the functions used to define expectations using EPT are also valid Turing models, i.e., replacing `@expectation` with `@model` yields a valid Turing program. Such a program corresponds to the density $\gamma_2(x) = \gamma(x)$ without requiring any transformation.

To create a Turing program corresponding to $\gamma_1^+(x)$ we need to multiply the density of the unaltered Turing program $\gamma(x)$ by $\max(f(x), 0)$. This is achieved using Turing's `@addlogprob!(z)` primitive, which increments the (log) density of the current execution by an arbitrary value `z`. Our program transformation is a pattern matching procedure that finds all the `return expr` statements in the function body and inserts a statement `@addlogprob!(log(max(expr, 0)))` before each one of them. A concrete example of the program transformation is presented in Figure 2. The transformation yielding density $\gamma_1^-(x)$ is analogous, but inserts a statement `@addlogprob!(log(-min(expr, 0)))` instead.

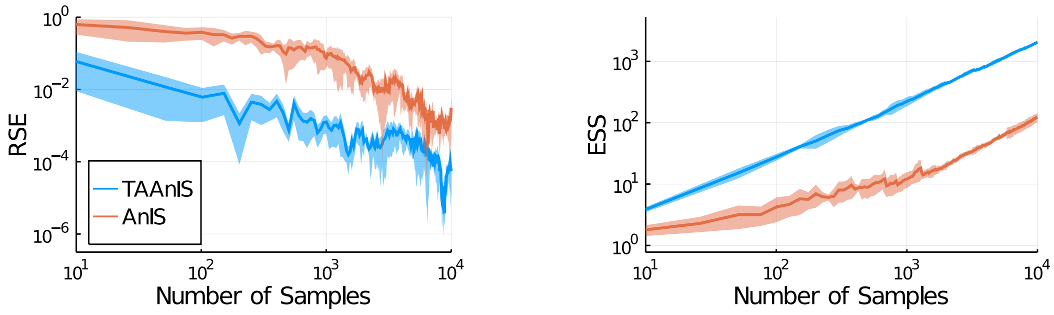

Figure 3: Relative squared error and effective sample size for posterior predictive experiments. The solid lines show the median of the estimator while the shaded region show the 25 % and 75 % quantiles. Medians and quantiles are computed over 10 separate runs with different random seed for the posterior predictive problem. For the ESS plot we are plotting $\min(\text{ESS}_{Z_1}, \text{ESS}_{Z_2})$ for TAAnIS and $\min(\text{ESS}_{\text{AnIS retargeted}}, \text{ESS}_{\text{AnIS}})$ for AnIS.

4. Experiments

While EPT is able to use any inference algorithm in Turing that provides marginal likelihood estimates we found that none of the existing inference algorithms in Turing is powerful enough for our use cases. For that reason we implemented a new inference engine for Turing which uses annealed importance samples (AnIS) (Neal, 2001). Appendix F gives a brief introduction to AnIS. AnIS requires the setting of two hyperparameters: an annealing schedule and a Markov chain Monte Carlo transition kernel. Currently, users can choose between the Metropolis-Hastings (MH) transition kernel which is implemented using Turing's `AdvancedMH.jl` (Turing Development Team, 2020) and the Hamiltonian Monte Carlo (HMC) transition kernel (Neal, 2011) which is built on `AdvancedHMC.jl` (Xu et al., 2020).

In our experiments we will use AnIS to refer to the non-target-aware setting of annealed importance sampling and target-aware annealed importance sampling (TAAnIS) to refer to the estimator that uses annealed importance sampling to separately estimate the marginal likelihoods described in Section 2. To ensure a fair comparison we use the same setting of hyperparameters for the AnIS algorithm as we will for the separate estimators in TAAnIS.

To compare the performance of the two estimators we look at the relative squared error (RSE) $\hat{\delta} := \frac{(\hat{\mu}-\mu)^2}{\mu^2}$, where μ denotes the ground truth value and $\hat{\mu}$ is the estimate, and the effective sample size (ESS). In our experiments we only consider target functions which are always positive, so from here on we use Z_1 to refer to Z_1^+ because we always have $Z_1^- = 0$. For TAAnIS we can look at the ESS of the two AnIS runs which we denote ESS_{Z_1} and ESS_{Z_2}. Similarly, ESS_{AnIS} is the ESS for AnIS. Note that we can also compute what the ESS would be if we would use the samples from AnIS to estimate Z_1 by multiplying the importance weights produced by AnIS by $f(x)$, we will denote the ESS computed in such a way by $\text{ESS}_{\text{AnIS retargeted}}$.

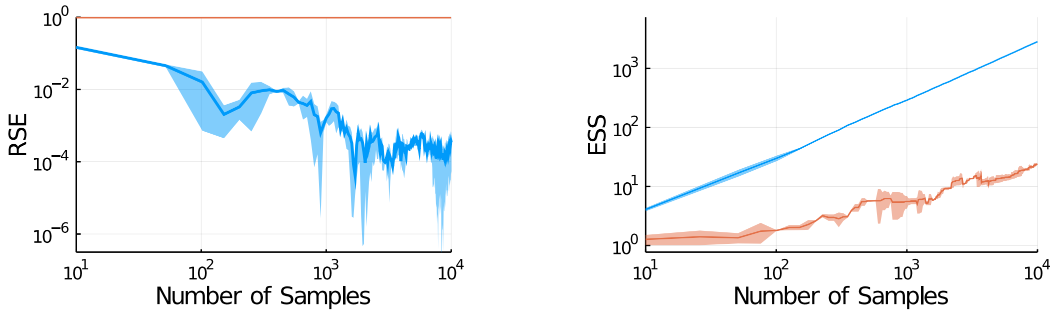

Figure 4: Relative squared error and effective sample size for SIR experiment. Conventions as in Figure 3. Results are computed over 5 runs for different random seeds.

4.1. Posterior Predictive

For our first experiment we compare AnIS and TAAnIS on the the task of calculating the posterior predictive distribution of a Gaussian model with unknown mean. Let

$$\gamma(\mathbf{x}) = \mathcal{N}(\mathbf{x}; 0, I)\,\mathcal{N}(\mathbf{y}; \mathbf{x}, I) \quad \text{and} \quad f(\mathbf{x}) = \mathcal{N}(-\mathbf{y}; \mathbf{x}, \frac{1}{2}I). \tag{2}$$

be our unnormalised density and target function. We assume our observed data is $\mathbf{y} = \frac{2}{\sqrt{10}}\mathbf{1}$ where $\mathbf{1}$ is a 10-dimensional vector of ones. The implementation of this model in Figure 1 highlights the user–friendly interface of EPT. The expectation can be defined in just 5 lines of code as per Figure 1. Figure 3 compares the performance of AnIS and TAAnIS. Overall we can see that there is a clear benefit of using the target-aware inference algorithm to estimate the expectation. TAAnIS achieves a lower RSE and the ESS analysis highlights the advantage of using separate marginal likelihood estimators for Z_1 and Z_2.

4.2. SIR Model

As a more applied example, assume we face an outbreak of a contagious disease. The government has provided us with a function yielding the expected cost of the disease which depends on the basic reproduction rate R_0.[1] To infer R_0 we choose to model the outbreak using a Susceptible-Infected-Recovered (SIR) model (Kermack et al., 1927). This model divides the population into three compartments: people who are susceptible to the the disease, people who are currently infected, and people who have already recovered. The dynamics of the outbreak are modelled by a set of differential equations which are governed by two parameters β and γ. Roughly, β models the constant rate of infectious contact between people, while γ is the constant recovery rate of infected individuals. From these parameters we can calculate the basic reproduction rate $R_0 = \frac{\beta}{\gamma}$. In our experiment, we assume γ to be fixed and we want to infer β and the initial number of infected people I_0. For a full description of the model and the experimental setup, see Appendix H.

This scenario is a good use case for EPT because we are interested in estimating a specific expectation with high accuracy. Furthermore, in this case there are some outcomes which

1. R_0 indicates the expected number of people one infected person will infect in a population in which all people are susceptible to the disease.

might be unlikely under the posterior but which incur a very high cost. By targeting only the posterior in our inference problem we run the danger of underestimating the probability of rare events which have a high cost and therefore underestimating the expected cost.

Figure 4 compares the performance of AnIS and TAAnIS for this problem. The RSE of the AnIS estimate fails to improve at all in the range of samples we collected. This is due to a strong mismatch between our target function and the posterior distribution. AnIS generates samples only in the regions with high posterior density but not in the regions where the target function is large. TABI methods were designed to overcome such a challenge.

Acknowledgments

We would like to thank Sheheryar Zaidi for helpful discussions on configuring Annealed Importance Sampling. Tim Reichelt and Adam Golinski are supported by UK EPSRC CDT in Autonomous Intelligent Machines and Systems.

References

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 0036-1445. doi: 10.1137/141000671.

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*, 20(28):1–6, 2019. ISSN 1533-7928.

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 F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314642.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690. PMLR, March 2018.

Adam Golinski, Frank Wood, and Tom Rainforth. Amortized Monte Carlo Integration. In *International Conference on Machine Learning*, pages 2309–2318. PMLR, May 2019.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, pages 167–181, New York, NY, USA, May 2014. Association for Computing Machinery. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593900.

Leo Grinsztajn, Elizaveta Semenova, Charles C. Margossian, and Julien Riou. Bayesian workflow for disease transmission modeling in Stan, 2020.

William Ogilvy Kermack, A. G. McKendrick, and Gilbert Thomas Walker. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 115(772):700–721, August 1927. doi: 10.1098/rspa.1927.0118.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic Inference by Program Transformation in Hakaru (System Description). In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming*, volume 9613, pages 62–79. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29603-6 978-3-319-29604-3. doi: 10.1007/978-3-319-29604-3_5.

Radford M. Neal. Annealed importance sampling. *Statistics and Computing*, 11(2):125–139, April 2001. ISSN 0960-3174. doi: 10.1023/A:1008923215028. URL https://doi.org/10.1023/A:1008923215028.

Radford M. Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, pages 113–162. Chapman & Hall / CRC Press, 2011.

Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An Efficient MCMC Sampler for Probabilistic Programs. *AAAI Conference on Artificial Intelligence (AAAI)*, page 7, 2015.

Tom Rainforth. *Automating Inference, Learning, and Design Using Probabilistic Programming*. http://purl.org/dc/dcmitype/Text, University of Oxford, 2017.

Tom Rainforth, Adam Golinski, Frank Wood, and Sheheryar Zaidi. Target–Aware Bayesian Inference: How to Beat Optimal Conventional Estimators. *Journal of Machine Learning Research*, 21(88):1–54, 2020.

Christian Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer-Verlag, New York, second edition, 2004. ISBN 978-0-387-21239-5. doi: 10.1007/978-1-4757-4145-2.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *2016 31st annual acm/ieee symposium on logic in computer science (lics)*, pages 1–10. IEEE, 2016.

Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

The Turing Development Team. TuringLang/AdvancedMH.jl. The Turing Language, October 2020. URL https://github.com/TuringLang/AdvancedMH.jl.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. *arXiv:1809.10756 [cs, stat]*, September 2018.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. In *Artificial Intelligence and Statistics*, pages 1024–1032. PMLR, April 2014.

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

Robert Zinkov and Chung-chieh Shan. Composing inference algorithms as program transformations. *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, page 10, 2017.

```
@model function model(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
end
```

Figure 5: Example Turing program.

Appendix A. Related Work

Nori et al. (2015) introduce a C-like probabilistic programming language with a semantics that defines the meaning of probabilistic programs as the expected value of its return expression. To construct a more efficient MCMC kernel, they generate a new program which is a transformation of the user-defined program. Their program transformations preserve the meaning of the original program i.e. it defines the same expected value. However, their end goal is to construct a more efficient inference algorithm *for the observed data* opposed to constructing a more efficient estimator *for the given target expectation*, which is the aim of our work.

Hakaru (Narayanan et al., 2016) provides the primitive `expect(m, f)` (Zinkov and Shan, 2017) which takes as input given a program defining a measure and a function returns a program which represents the expectation over the measure with respect to the function. However, this primitive is mainly used to get a symbolic representation of the expectation with the primary aim to compute the normalisation constant of unnormalised measures. measures e.g. when conditioning a joint probability distribution on data. With the help of a computer algebra solver the symbolic representation can be used to compute the integral analytically.

To our knowledge, our framework is the first to use the semantics of probabilistic programs defining expectations over their return expressions to generate *target-aware inference algorithms*.

Appendix B. A Short Introduction to Turing

We will present a brief overview of the Turing PPL and its internals. For more details we refer the reader to the Turing documentation.[2] A Turing program is defined similarly to a normal Julia (Bezanson et al., 2017) function: the `@model` macro indicates the definition of a Turing model which means we can use tilde statements to define random variables inside the function body e.g., `x ~ Normal(0, 1)` (see Figure 5). The left-hand side of the tilde statement must be a random variable, and the right-hand side must be a distribution. The values of the observed random variables must be passed as arguments to the function.

The `@model` macro takes the function and creates a Turing model generator with the same name as the function. If we call this model generator with some specific function arguments we get back a Turing model. At that stage the tilde statements get resolved into calls to the primitives `tilde_assume` or `tilde_observe`. The exact behaviour of those primitives usually depends on the inference algorithm that is used. Here we will describe

2. https://turing.ml/dev/docs/using-turing/

the behaviour of `tilde_assume` and `tilde_observe` when we want to evaluate the density of a raw random draw of the program.

The primitive `tilde_observe(ctx, sampler, dist, value, vname, vinds, vi)` gets called if the left-hand side of the tilde statement was part of the observed data (i.e. one of the formal parameters of the function). `dist` will be a distribution, `value` will be a value in the support of that distribution, `vi` is a data structure which keeps track of the sampled values and the log density of the current execution and `vname` is the random variable identifier i.e. the lexical address that is used to uniquely identify the random variable. For the purpose of this document the other function arguments can be ignored. Mainly, `tilde_observe` evaluates the log density of `value` under `dist` and adds the value to the log density stored in `vi`.

If the left-hand side of the tilde statement is not part of the observed data then the primitive `tilde_assume(rng, ctx, sampler, dist, vname, inds, vi)` gets called. `dist`, `vi` and `vname` have the same types as in the `tilde_observe` call. On a high-level `tilde_assume` samples a value from `dist` and then stores the sampled value in `vi` and adds the corresponding log density to the log density stored in `vi`.

Building on the notation of Rainforth (2017, Chapter 4.3) we can formalise the definitions of `tilde_assume` and `tilde_observe`. `tilde_assume` takes as input a distribution which is constructed from a set of variables η, formally, $g_a(x|\eta)$ where a is the lexical address of the `tilde_assume` statement. Similarly `tilde_observe` takes as input a distribution constructed from a set of variables ϕ and some observed random variables y, or formally, $h_b(y|\phi)$ where b again is a lexical address.

Additionally, Turing also provides the `@addlogprob!(x)` primitive which increments the log density of the current execution by an arbitrary value `x`. We let $\psi_1, \ldots, \psi_K$ denote all the terms that are added to the density using `@addlogprob!`.

Given the above definitions the density of a Turing program is

$$\gamma(x) = \prod_{i=1}^{n_x} g_{a_i}(x_i|\eta_i) \prod_{j=1}^{n_y} h_{b_j}(y_j|\phi_j) \prod_{k=1}^{K} \exp(\psi_k).$$

Therefore the program in Figure 5 defines the density $\gamma(x) = \mathcal{N}(x; 0, 1)\,\mathcal{N}(y; x, 1)$ where y is the observed data. Note that there are no `@addlogprob!(x)` statements in this program.

Appendix C. A Note On the Formalization of $f(\cdot)$

To provide a valid input for our normalization constant estimators in the TABI framework, we need to be able to provide a valid density. This also means we need to be careful about the random variable this density is defined over. When we construct, e.g., $\gamma(x)^+$ we are effectively including an additional factor $f^+(x)$ to the program density (informally, an additional likelihood term). As such, this density is defined with respect to the raw random draws of the program, as per a conventional probabilistic program (see e.g. (Rainforth, 2017, Section 4.3)). This means that the formal definition of $f(\cdot)$ is the full mapping from the raw random draws to the returned values, rather than what is lexically written after the `return` statement(s). This is why, for example, it is still valid to have multiple different `return` statements in program, provided these all return outputs of the same type and dimensionality.

In practice, this is not something we need to worry about when writing either models or inference engines, as the law of the unconscious statistician protects us from needing to carefully delineate between what our function is and what our random variable is. Essentially, the output of the program is a random variable and we are calculating the expectation of this random variable; this expectation does not vary if we change the parameterization of our model. However, the distinction is important from the perspective of ensuring that the approach is valid, both in terms of providing valid inputs to the individual TABI estimators, and in terms of ensuring the inference engines are valid for these problems.

Appendix D. Estimating Expectations in Turing

```
@model function model(y)
    x ~ MvNormal(zeros(length(y)), I)
    y ~ MvNormal(x, I)
end

D = 10
y_observed = -2 * ones(D) / sqrt(D)

num_samples = 1000
posterior_samples = sample(
    model(y_observed), NUTS(0.65), num_samples)

posterior_x = Array(posterior_samples[:x])
expectation_estimate = mean(
    map(x -> pdf(MvNormal(x, 0.5*I), -y_observed), posterior_x))
```

Full example of the estimation of an expectation with the Turing language. The user first defines the model, then conditions it on some observed data, computes posterior samples and then uses these samples to compute a Monte Carlo estimate of the expectation.

Appendix E. Full Example of `@expectation` Macro Transformation

The expectation

```
@expectation function expct(y)
    x ~ MvNormal(zeros(length(y)), I)
    y ~ MvNormal(x, I)
    return pdf(MvNormal(x, 0.5*I), -y)
end
```

gets transformed into

```
@model function gamma1_plus(y)
    x ~ MvNormal(zeros(length(y)), I)
    y ~ MvNormal(x, I)
    tmp = pdf(MvNormal(x, 0.5*I), -y)
    @addlogprob!(log(max(tmp, 0)))
    return tmp
end

@model function gamma1_minus(y)
    x ~ MvNormal(zeros(length(y)), I)
    y ~ MvNormal(x, I)
    tmp = pdf(MvNormal(x, 0.5*I), -y)
    @addlogprob!(log(-min(tmp, 0)))
    return tmp
end

@model function gamma2(y)
    x ~ MvNormal(zeros(length(y)), I)
    y ~ MvNormal(x, I)
    return pdf(MvNormal(x, 0.5*I), -y)
end

expct = Expectation(
    gamma1_plus,
    gamma1_minus,
    gamma2
)
```

The type `Expectation` is simply used to have one common object which stores the three different Turing models. Notice that for `gamma2` the function body is identical to the original function.

Appendix F. Annealed Importance Sampling

Annealed importance sampling (AnIS) (Neal, 2001) is an inference algorithm which was developed with the goal of efficiently estimating the normalisation constant Z of an unnormalised density $\gamma(x)$. It works by defining a sequence of annealing distributions $\pi_0(x), \dots, \pi_n(x)$ which anneal between a simple (i.e. we can sample from it and evaluate its density) base distribution $\pi_0(x)$ and the complex target density $\pi_n(x) = \gamma(x)$. The most common scheme for defining the annealing distributions is to set them to $\pi_i(x) \propto \lambda_i(x) = p(x)^{1-\beta_n}\gamma(x)^{\beta_n}$, with $p(x)$ being the prior distribution and $0 = \beta_0 < \beta_1 < \cdots < \beta_n = 1$. Furthermore, the algorithm requires the definition of Markov chain transition kernels $\tau_1(x, x'), \dots, \tau_{n-1}(x, x')$. Each $\tau_i(x, x')$ needs to leave its target distribution $\lambda_i(x)$ invariant.

The algorithm to create a j^{th} single weighted sample then becomes:

1. Sample initial particle $x_j^{(1)} \sim \pi_0(x)$

2. For $i = 1, \dots, (n-1)$: Generate $x_j^{(i+1)} \sim \tau_i(x_j^{(i)}, \cdot)$

3. Return sample $x_j^{(n)}$ with weight

$$w_j = \frac{\lambda_1(x_j^{(1)})\lambda_2(x_j^{(2)})\dots\lambda_n(x_j^{(n)})}{p(x_j^{(1)})\lambda_1(x_j^{(2)})\dots\lambda_{n-1}(x_j^{(n)})}$$

We can estimate expectations with the weights and samples just as in standard importance sampling. Thus an estimate for the normalisation constant is

$$Z \approx \frac{1}{N}\sum_{j=1}^{N} w_j,$$

and an estimate for the expectation is

$$\mathbb{E}_{\pi(x)}[f(x)] \approx \frac{\sum_{j=1}^{N} w_j f(x_j^{(n)})}{\sum_{j=1}^{N} w_j}.$$

Appendix G. Hyperparameters for Experiments

For both problems our target function is always positive which means in the TABI estimator we only have to compute Z_1^+ and Z_2 because $Z_1^- = 0$. Therefore, target-aware annealed importance sampling (TAAnIS) runs standard annealed importance sampling twice, one time to estimate Z_1^+ and the other time to estimate Z_2. For the experiments, on the same problem we always use the same hyperparameters for the annealed importance sampling algorithm both to run AnIS and for the two estimates in TAAnIS.

G.1. Posterior Predictive

For the annealed importance sampling we use a MH transition kernel with a standard normal as the proposal and 20 MH steps on each annealing distribution. We use 100 annealing distributions uniformly spaced.

G.2. SIR Model

For the annealed importance sampling estimators we use HMC transition kernels with a step size of 0.05, 10 leapfrog steps and 10 MCMC steps on each annealing distribution. We use 100 geometrically spaced annealing distributions.

The ground truth is computed using importance sampling with 10^8 samples and the prior as a proposal distribution. See Equation (4) for the full SIR model including the priors.

Appendix H. SIR Experiment

The SIR model assumes the population is divided into three compartments: susceptible (people not yet infected), infected and recovered (people who already had the disease). Apart from a number of initially infected people I_0, every person in the population starts out as susceptible, then potentially becomes infected and then recovers after some time.

Crucially, the SIR model makes some simplifying assumptions most notably that a recovered person cannot get infected again and that the total number of people in the population remains constant.

The SIR model can be succinctly described by the following set of ordinary differential equations (ODE):

$$\frac{dS}{dt} = -\beta S \frac{I}{N} \tag{3a}$$

$$\frac{dI}{dt} = \beta S \frac{I}{N} - \gamma I \tag{3b}$$

$$\frac{dR}{dt} = \gamma I \tag{3c}$$

For our case we assume that $\gamma = 0.25$ is known and need not be inferred from data. Therefore our model has two unknown parameters: the initial number of infected patients I_0 and β. We assume we are given data in the form of observations y_i, the number of observed newly infected people on day i. This gives us the statistical model

$$\gamma = 0.25, \tag{4a}$$

$$\beta \sim \text{truncated}(\text{Normal}(2, 1.5), 0, \infty), \tag{4b}$$

$$i_0 \sim \text{truncated}(\text{Normal}(10, 10), 0, 1000), \tag{4c}$$

$$I_0 = i_0 * 10; S_0 = 10000 - I_0; R_0 = 0, \tag{4d}$$

$$\mathbf{x} = \text{ODESolve}(\beta, \gamma, S, I, R), \tag{4e}$$

$$y_i \sim \text{NegBinomial}(x_i, 0.5). \tag{4f}$$

Here ODESolve indicates a call to an ODE solver which solves the set of equations (3). It outputs x_i the predicted number of newly infected people on day i. Since we assume that we have noisy observations we use a negative Binomial to add some noise to the predictions from the differential equations. The negative binomial NegBinomial(μ, ϕ) is parametrised by its mean μ and an overdispersion parameter ϕ; for $\phi \to \infty$ the negative binomial distribution approaches a Poisson distribution. For an in-depth discussion about doing Bayesian parameter inference in the SIR model we refer the reader to the case study of Grinsztajn et al. (2020).

We assume we are given a cost function in terms of the basic reproduction rate,

$$\text{cost}(R_0) = 1e12 * \text{logistic}(10R_0 - 30).$$