Adapting Probabilistic Programming Systems to Estimate Expectations Efficiently

Abstract

We show that the standard computational pipeline of probabilistic programming systems (PPSs) can be inefficient for estimating expectations and introduce the concept of *expectation programming* to address this. In expectation programming, the aim of the backend inference engine is to directly estimate expected return values of programs, as opposed to approximating their conditional distributions. This distinction, while subtle, allows us to achieve substantial performance improvements over the standard PPS computational pipeline by tailoring computation to the expectation we care about. We realize a particular instance of our expectation programming concept, Expectation Programming in Turing (EPT), by extending the PPS *Turing* to allow so-called *target-aware* inference to be run automatically. We then verify the statistical soundness of EPT theoretically, and show that it provides substantial empirical gains in practice.

1 INTRODUCTION

Estimating *expectations* is at the center of many scientific workflows. For example, the decision theoretic foundations of most statistical paradigms, e.g. Bayesian decision theory, are rooted in calculating the expectation of a loss function [Robert and Casella, 2004].

Carrying out this estimation often requires *approximate inference* to be performed: we may not be able to directly draw samples of the random variable we wish to calculate the expectation of, or a simple Monte Carlo estimate might produce problematically high variance.

Probabilistic programming systems (PPSs) provide a powerful basis for encoding such inference problems and then assisting with, or even fully automating, the approximation of their solution [Gordon et al., 2014, van de Meent et al., 2018]. In a PPS, programs are typically specified (often indirectly) through an unnormalized density $\gamma(x)$. Assuming analytic solutions are not available, the role of the system's inference engine is now to construct an approximation, $\hat{\pi}(x)$, for the distribution specified by the normalized density $\pi(x) = \gamma(x)/Z$, where Z is an unknown normalizing constant and $\pi(x)$ typically represents a conditional distribution, such as the posterior in a Bayesian modeling setting. This approximation can then be used in turn for downstream tasks, such as approximating one or more expectations.

Though ostensibly very general, our key insight is that this standard PPS computational pipeline—which is implicitly followed by all contemporary PPSs that conduct inference approximately (e.g. 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], Mansinghka et al. [2014], Goodman and Stuhlmüller [2014], Murray and Schön [2018], Minka et al. [2018])—can be highly suboptimal when our ultimate aim is to estimate a particular expectation, $\mathbb{E}_{\pi(x)}[f(x)]$. This is because such a pipeline fails to perform estimation in a *target-aware* fashion: it does not allow information about f to be exploited by the inference engine, thereby forgoing the substantial empirical gains that using information about f can yield [Torrie and Valleau, 1977, Hesterberg, 1988, Wolpert, 1991, Oh and Berger, 1992, Evans et al., 1995, Meng and Wong, 1996, Chen et al., 1997, Gelman and Meng, 1998, Lacoste-Julien et al., 2011, Owen, 2013, Golinski et al., 2019, Rainforth et al., 2020]. Note here that it is not generally possible to incorporate the required information about f by adjusting the model definition; fundamental changes to the computational pipeline itself are required.

To address this, we introduce, and formalize, the concept of *expectation programming*. Here an expectation program is analogous to a probabilistic program, but its target quantity of interest is the expected value of the program's return values, rather than their conditional distribution. This subtle distinction leads to changes in the requirements for the pro-

Submitted to the 38th Conference on Uncertainty in Artificial Intelligence (UAI 2022). **To be used for reviewing only**.

gram to be valid, and, critically, the estimation that must be performed by the backend inference engine. This, in turn, allows us to construct computational pipelines which are target-aware, utilizing information in the program itself to estimate expectations substantially more efficiently than can be achieved by existing PPSs.

We realize our expectation programming concept through a specific system we call **EPT** (Expectation Programming in Turing), built upon the Turing PPS [Ge et al., 2018]. EPT takes as input a Turing-style program and uses a combination of program transformations and existing inference strategies to construct target-aware estimators via the TABI approach of [Rainforth et al., 2020].

We formally demonstrate the statistical soundness of EPT, proving that it produces consistent estimates under nominal assumptions. We further show empirically that it can be used to express and run effective inference for a number of problems, finding that it produces estimates that are significantly more accurate than conventional usage of Turing. As part of this, we also implement a new annealed importance sampling (AnIS) [Neal, 2001] inference engine for Turing, finding that this allows for effective marginal likelihood estimation in a much wider array of problems than Turing's previously supported inference strategies.

To summarize, our key contributions are: a) identifying the shortfall of existing PPSs when estimating expectations and introducing the concept of expectation programming to address this; b) developing EPT as a particular realization of the expectation programming concept; c) formalizing the notion of an expectation program and demonstrating the statistical correctness of EPT; d) introducing a new AnIS inference engine to Turing; and e) showing that EPT can provide substantial empirical benefits over conventional use of Turing on real problems.

2 BACKGROUND

2.1 TURING PROGRAMS AS DENSITIES

To provide a basis for introducing expectation programming, we consider the PPS Turing (Ge et al. [2018], `https://turing.ml/dev/docs/using-turing/`), but note that the concepts introduced apply to PPSs in general. We provide a brief introduction to Turing here, along with our own new formalism for the densities Turing program define by extending the approach of Rainforth [2017, §4.3]. This is necessitated by some technical intricacies of the expectation programming approach. To assist with this, we will use the following simple Turing program as a running example:

```
@model function model(y)
    x ~ Normal(0, 1)
    @addlogprob!(0.1)
    y ~ Normal(x, 1)
end
```

A Turing program is defined similarly to a normal Julia function [Bezanson et al., 2017]: the `@model` macro indicates the definition of a Turing model, with tilde statements inside the body, e.g. `x ~ Normal(0, 1)`, to denote probabilistic model components. Observed data can be passed in as a formal argument to the function. If the variable name on the left-hand side of the tilde statement is not part of the arguments of the functions then it is interpreted as a random variable.

Let $x_{1:n}$ denote the set of direct outputs from sampling statements and $y_{1:m}$ the observed data. We can view Turing programs as defining an unnormalized density $\gamma(x_{1:n})$ (with an implicit appropriate reference measure). To compute the density for a given $x_{1:n}$ the program executes like a normal Julia program, while keeping track of the density of the current execution. Specifically, when Turing reaches a tilde statement corresponding to a random variable, it samples a value for x_i, evaluates the density of this draw, and factors this into the overall execution density. We denote the density of the draw as $g_i(x_i|\eta_i)$, where g_i denotes the form of the sampling statement and η_i its parameters. For the tilde statements corresponding to the observed data, it evaluates the density function $h_j(y_j|\phi_j)$—where h_j and ϕ_j are analogous to g_i and η_i respectively—and factors the overall density accordingly.

Sometimes a user might want to add additional factors to the density without using a tilde statement. For this, Turing provides the `@addlogprob!(log_p)` primitive which multiplies the density of the current execution by an arbitrary value `exp(log_p)`. We use $\psi_1, \ldots, \psi_K$ to denote all the terms that are added to the density using `@addlogprob!`.

Putting these together, the unnormalized density defined by any valid program trace can be written as

$$\gamma(x_{1:n}) = \prod_{i=1}^{n} g_i(x_i|\eta_i) \prod_{j=1}^{m} h_j(y_j|\phi_j) \prod_{k=1}^{K} \exp(\psi_k). \quad (1)$$

Our example program thus defines the density $\gamma(x) = \exp(0.1)\mathcal{N}(x;0,1)\mathcal{N}(y;x,1)$, with a fixed input y. Note here that everything (i.e. $n, x_{1:n}, \eta_{1:n}, g_{1:n}, m, y_{1:m}, \phi_{1:m}, h_{1:m}, K, \psi_{1:K}$) can be a random variable because of potential stochasticity in the program path. However, using the program itself, everything is deterministically calculable from $x_{1:n}$, which can thus be thought of as the 'raw' random draws that dictate all the randomness of the program; everything else is a pushforward of these.

2.2 TARGET-AWARE INFERENCE

Consider the problem of estimating an expectation 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 a known unnormalized density, but Z

is an unknown normalization constant (sometimes referred to as the marginal likelihood).

The inference engines in PPSs like Turing are setup to approximate $\pi(x)$ of this form. As such, the standard pipeline to approximate an expectation using a PPS is to first approximate $\pi(x)$ (e.g. with samples) and then use this to approximate the expectation in turn.

Unfortunately, this ignores information about f and is therefore suboptimal if f is known [Golinski et al., 2019]. While one might initially expect that information about f can be easily incorporated through simple model adjustments, this is unfortunately not the case in practice: any adjustments we make will mean we need to estimate an additional corrective factor on top of performing inference for the new model. Indeed, naive approaches to incorporating information about f, like adding $|f(x)|$ as a density factor to the model, have been found to typically worsen, rather than improve, the final estimates [Rainforth et al., 2020].

Rainforth et al. [2020] recently showed that this issue stems from fundamental limitations of the efficacy of using a *single* Monte Carlo estimator for such expectations. Namely, through their Target-Aware Bayesian Inference (TABI) framework, they show that by breaking down the expectation into three parts:

$$\mathbb{E}_{\pi(x)}[f(x)] = (Z_1^+ - Z_1^-)/Z_2, \tag{2}$$

where $Z_1^+ = \int \gamma(x) f^+(x) dx$, $Z_1^- = \int \gamma(x) f^-(x) dx$, $Z_2 = \int \gamma(x) dx$, $f^+(x) = \max(f(x), 0)$, and $f^-(x) = -\min(f(x), 0)$, and then estimating each term separately, one can often achieve a substantially improved overall estimator, $\mathbb{E}_{\pi(x)}[f(x)] \approx (\hat{Z}_1^+ - \hat{Z}_1^-)/\hat{Z}_2$.

The intuition here is that each individual term can often be estimated more accurately in isolation than the original expectation. To see this, first note that the three subcomponents can be seen as the respective normalization constants of the three densities

$$\begin{aligned} \gamma_1^+(x) &\propto \gamma(x) f^+(x), \\ \gamma_1^-(x) &\propto \gamma(x) f^-(x), \\ \gamma_2(x) &= \gamma(x). \end{aligned} \tag{3}$$

The TABI framework now allows one to define a separate estimator *tailored* to each of these problems. In general, it allows one to repurpose any algorithm which provides estimates of the normalization constant into a target-aware inference algorithm by separately applying it to each of $\gamma_1^+(x)$, $\gamma_1^-(x)$, and $\gamma_2(x)$. TABI can theoretically achieve an arbitrarily low error for any fixed sample budget (≥ 3), unlike standard approaches such as self-normalized importance sampling or MCMC whose expected error is lower bounded, even when using an optimal proposal/sampler. The achievable gains increase, both theoretically and empirically, with the degree of mismatch between $\pi(x)$ and $\pi(x)f(x)$.

3 EXPECTATION PROGRAMMING

At a high level, *expectation programming* adapts probabilistic programming systems to automate the estimation of expectations in a *target-aware* manner. As we now explain, an *expectation program* is analogous to a probabilistic program, but where the quantity of interest is the expectation of its return values under the program's conditional distribution, rather than the conditional distribution itself.

3.1 FORMALIZATION

To formalize the concept of an expectation program, we first statistically formalize probabilistic programs as follows.

Definition 1. *A probabilistic program $\mathcal{P}$ in a probabilistic programming language defines an unnormalized density $\gamma(x_{1:n})$ over the raw random draws $x_{1:n} \in \mathcal{X}$ of the program, which collectively we refer to as the program trace, along with an implicitly defined reference measure μ.*

We let $\pi(x_{1:n}) = \gamma(x_{1:n})/Z$ denote the normalized density with $Z = \int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n})$. Here $\pi(x_{1:n})$ and μ combined implicitly define the conditional probability distribution specified by $\mathcal{P}$, which we denote $\mathbb{P}(A) = \int_A \pi(x_{1:n}) d\mu(x_{1:n})$.

To ensure that the induced probability measure of a program is well-defined, we require that $\gamma(x_{1:n})$ corresponds to a valid unnormalized density. This guarantees that there is a valid probability distribution the inference algorithm of the particular PPS can converge to. We use this to formalize the concept of a *valid* probabilistic program as follows.

Definition 2. *A probabilistic program, $\mathcal{P}$, is* valid *(and defines a valid unnormalized probabilistic program density $\gamma(x_{1:n})$) if and only if both of the following hold: $\gamma(x_{1:n}) \geq 0, \forall x_{1:n} \in \mathcal{X}$; and $0 < \int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n}) < \infty$.*

For Turing we have described how programs specify $\gamma(x_{1:n})$ in Section 2.1, but Definitions 1 and 2 apply more generally and only require that we can derive an unnormalized density function for a given program; a requirement that is satisfied by most existing popular PPSs.

We can now formalize the concept of an expectation program by associating return values to our program:

Definition 3. *An expectation program, $\mathcal{E}$, is a probabilistic program (as per Definition 1) with an associated set of return values $F \in \mathcal{F} \subseteq \mathbb{R}^d$ that are a deterministic mapping of the trace $x_{1:n}$.*

From this definition we see that expectation programs are largely equivalent to probabilistic programs, indeed programs in any PPS that allows return values will also be expectation programs provided their outputs are numeric and fixed dimensional. However, as their underlying quantity of interest is the expectation of their return values, $\mathbb{E}[F]$,

they require a slightly different set of assumptions to ensure validity as follows.

Definition 4. *An expectation program $\mathcal{E}$ is valid if and only if it is a valid probabilistic program and F is integrable.*

Here the additional requirement of the expectation program's outputs being integrable essentially equates to requiring that the expectation $\mathbb{E}[F]$ exists and $\mathbb{E}[|F_i|] < \infty$ for each dimension F_i of F. This is generally a very weak requirement, and strictly weaker than an assumption typically implicitly made by existing PPSs when confirming the validity of their inference engines as discussed in Appendix B.

To link expectation programs back into our early expectation notation, we now note that the requirement for the return values to be a deterministic mapping of the trace means that we can write $F = f(x_{1:n})$, such that $\mathbb{E}[F] = \mathbb{E}_{\pi(x_{1:n})}[f(x_{1:n})]$. Thus the formal definition of the function we are taking the expectation of is that it is the full mapping from the raw random draws to the returned values rather than what is lexically written in any **return** statement(s). This is why, for instance, it is still valid to have multiple different **return** statements in a program; provided each **return** statement defines the same number of return values. 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 relieves us from explicitly delineating the random variable defined by our function (the expectation of this random variable does not vary if we change the parameterization of our model). However, the distinction is important for ensuring validity and to identify the precise target function we wish to extract information about when making the inference target-aware.

3.2 TARGET-AWARE INFERENCE ENGINES

The key idea of our expectation programming paradigm is to use the formalisms from the previous section to set up inference engines that exploit information from f to perform target-aware estimation. As explained in Section 2.2, this can lead to estimators that provide substantial performance improvements over the standard PPS approach of simply approximating $\pi(x_{1:n})$, ignoring $f(x_{1:n})$ completely.

Note that the approximate computation we are performing here is fundamentally different to that of conventional inference engines: we are estimating an expectation, rather than approximating a conditional distribution. This means the form of the outputs from our engine will change, while we will have to exploit additional information about the program. As such, we will generally need to make changes to how the program itself is processed, rather than just implementing a new inference engine in the existing PPS structure. Thankfully though, it will still usually be possible to repurpose existing inference engines as part of an overall target-aware estimation scheme, as we now show.

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)                # x ~ N(x; 0, 1)
    y ~ Normal(x, 1)                # y ~ N(y; x, 1)
    return x^3                      # f(x) = x^3
end
expct_estimate, diagnostics =
  estimate_expectation(expt_prog(2),
    TABI(marginal_likelihood_estimator =
      TuringAlgorithm(AnIS(),num_samples=100)))
```

Figure 1: An example of estimating an expectation with EPT. Here **estimate_expectation** is our "do estimation" call which takes in expectation program `expt_prog` (with input $y = 2$) and an estimation method to apply (here a TABI estimator using annealed importance sampling), and returns an estimate for the expected return value of `expt_prog`.

3.3 EXPECTATION PROGRAMMING IN TURING

We now introduce a particular realization of the expectation programming concept which we call *Expectation Programming in Turing* (EPT). EPT builds on the PPS Turing to provide a highly effective, and surprisingly simple, mechanism to perform expectation programming. It allows users to specify $\gamma(x)$ analogously to how they would using Turing's `@model` macro, and uses Turing's **return** semantics to define F and thus $f(x)$.

The key component of the EPT is splitting up the estimation of the desired expectation as per the TABI framework of Section 2.2. To do so we use source-code transformations to generate three different Turing programs, one for each of the densities $\gamma_1^+(x)$, $\gamma_1^-(x)$, and $\gamma_2(x)$ (as per Equation (3)). We then estimate the expectation by individually estimating the normalization constant of each of these densities and then combining them as per Equation (2). Generating valid Turing programs allows us to leverage any inference algorithm in Turing that provides marginal likelihood estimates to estimate the quantities Z_1^+, Z_1^-, and Z_2. This modularity means that we do not have to implement custom inference algorithms that would only work with EPT.

Estimating expectations with EPT is done in two stages. First, users define an expectation program with the `@expectation` macro, which is a drop-in replacement for `@model`, and an example for which is shown in Figure 1. Using code transformations, `@expectation` automatically generates the three Turing programs representing the densities $\gamma_1^+(x)$, $\gamma_1^-(x)$, and $\gamma_2(x)$. This happens behind the scenes and the user does not need to deal with the transformed programs directly.

To estimate the expectation, the user then calls **estimate_expectation**`(expt_prog, method)`, where `method` specifies the estimation approach to be used. At present, the only supported class of methods is **TABI**, which implements the previously explained TABI estimators, but the syntax is designed to allow for easy addition of hypothetical alternative approaches.

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end
```

```
@model function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    @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. The full transformation is slightly more complex due to Turing's internals. Appendix D shows the full source code transformation for this model.

EPT then estimates the normalization constants Z_1^+, Z_1^-, and Z_2 by running a Turing inference algorithm on each Turing program generated by `@expectation` and combining the normalization constant estimates to form an estimate of the expectation. In the example in Figure 1, we use **`TABI`** with annealed importance sampling **`AnIS`**, which is a new Turing inference algorithm that we have added to the system for the purposes of this paper. **`TuringAlgorithm`** is a thin-wrapper object storing the necessary information that allows **`TABI`** to use a Turing inference method. **`AnIS`** can be substituted with any other Turing inference algorithm that returns a marginal likelihood estimate. Here **`AnIS`**`()` implies the use of some arbitrary default AnIS parameters regarding the Markov chain transition kernel, and the number and spacing of intermediate potentials used.

3.4 PROGRAM TRANSFORMATIONS

We now consider how to generate the Turing programs corresponding to each of the TABI densities. Note that expectation programs in EPT are also valid Turing models, i.e., replacing `@expectation` with `@model` yields a valid Turing program. Such a program corresponds to the unnormalized density $\gamma_2(x) = \gamma(x)$ without requiring any transformation of the source-code.

To create a Turing program corresponding to $\gamma_1^+(x)$, we need to multiply the unnormalized density of the unaltered Turing program $\gamma(x)$ by $\max(f(x), 0)$. This is achieved using Turing's aforementioned `@addlogprob!` primitive, such that we can think of it as adding a new factor $\max(f(x_{1:n}), 0)$ to the program density definition in (1). Our transformations are pattern matching procedures that find all the **`return`** `expr` statements in the function body and then a) create a new local variable `tmp = expr` (where `tmp` is a unique identifier generated using `gensym()`), b) insert a statement `@addlogprob!(log(max(tmp, 0)))` before the **`return`**, and c) change the return statement itself to **`return`** `tmp`. A concrete example of the transformation is presented in Figure 2. The transformation for $\gamma_1^-(x)$ is analogous but inserts a statement `@addlogprob!(log(-min(tmp, 0)))` instead.

Users can define multiple expectations by specifying multiple return values, while each individual return value needs to almost surely be a numerical scalar. This ensures that each target expectation is well defined and individually identified. For each return expression, we apply our program transformation separately and derive a corresponding TABI estimator for each. For example, if we have **`return`** `expr1, expr2, expr3`, the program transformation for $\{\gamma_1^+(x)\}_2$ would add the statement `@addlogprob!(log(max(expr2, 0)))`. Appendix I shows a full example of this.

3.5 VALIDITY OF EPT

We now formalize and demonstrate the statistical correctness of the EPT approach. For simplicity, we will assume throughout that programs almost surely return a single scalar value (i.e. the probability that the return value fails to be a well-defined scalar is 0). Generalization to programs with multiple return values is straightforward (provided the number of return values is fixed) by considering each return value separately in isolation (as EPT does itself).

Theorem 1. *Let $\mathcal{E}$ be a valid expectation program in EPT with unnormalized density $\gamma(x_{1:n})$, defined on possible traces $x_{1:n} \in \mathcal{X}$, with return value $F = f(x_{1:n})$. Then $\gamma_1^+(x_{1:n}) := \gamma(x_{1:n}) \max(0, f(x_{1:n}))$, $\gamma_1^-(x_{1:n}) := -\gamma(x_{1:n}) \min(0, f(x_{1:n}))$, and $\gamma_2(x_{1:n}) := \gamma(x_{1:n})$ are all valid unnormalized probabilistic program densities. Further, if $\{\hat{Z}_1^+\}_m$, $\{\hat{Z}_1^-\}_m$, $\{\hat{Z}_2\}_m$ are sequences of estimators for $m \in \mathbb{N}^+$ such that*

$$\{\hat{Z}_1^\pm\}_m \xrightarrow{p} \int_{\mathcal{X}} \gamma_1^\pm(x_{1:n}) d\mu(x_{1:n}),$$

$$\{\hat{Z}_2\}_m \xrightarrow{p} \int_{\mathcal{X}} \gamma_2(x_{1:n}) d\mu(x_{1:n})$$

where $\xrightarrow{p}$ means convergence in probability as $m \to \infty$, then $(\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m)/\{\hat{Z}_2\}_m \xrightarrow{p} \mathbb{E}[F]$.

Theorem 1, which is proved in Appendix B, shows that if we have programs with the desired densities and we use consistent marginal likelihood estimators for each, then our resulting expectation estimates will themselves be consistent. The latter is covered by the consistency of Turing's own inference engines. The former requires that our transformed programs are valid Turing programs with the intended densities. We now show that this is indeed the case.

Given an input EPT program $\mathcal{E}$, EPT applies transformations to get the three Turing programs $\mathcal{P}_1^+$, $\mathcal{P}_1^-$, and $\mathcal{P}_2$ with $\gamma_1^+(x_{1:n})$, $\gamma_1^-(x_{1:n})$, and $\gamma_2(x_{1:n})$ as their respective densities. To ensure that the transformations for $\gamma_1^+(x_{1:n})$ and $\gamma_1^-(x_{1:n})$ are correct, we need to ensure that a) the inserted code in our transformations is itself valid, b) the transformation does not have any unintended side effects, and c) the new density terms add valid factors to the program density. The first is true as the operation of the transformed sections of code are identical to the originals except for the new `@addlogprob!` terms, which themselves produce no outputs and, by construction, use only the variables that are in scope. The second is guaranteed by ensuring that the `tmp` variables are given unique identifiers that cannot clash with each other or any other variables in the program. The third follows from the restriction that each return value must almost surely be a numerical scalar, coupled with the fact that the added density factors (namely `max(tmp, 0)` and `-min(tmp, 0)`) are non-negative by construction.

Thus, we have shown that EPT will produce a consistent estimation of program expectations, under the assumptions of Definition 4 and the consistency of the base inference algorithms implemented in Turing.

4 RELATED WORK

Our focus is explicitly on the case of *estimating* expectations. Though a few papers [Gordon et al., 2014, Zinkov and Shan, 2017] have provided alternative formalizations for the expectation defined by a probabilistic program, none do this from the perspective of directly targeting this expectation as the quantity to estimate. Relatedly, a few languages provide primitives to compute expectations *analytically* in the rare situation where this is possible, such as Hakaru [Zinkov and Shan, 2017] or λPSI [Gehr et al., 2020]. Unlike in our setting, these do not require notable changes to the backend computation from the standard inference setting because the underlying problem remains the same: calculate an integral analytically. The contributions of these works are thus somewhat tangential to our own, with our key message being that *estimating* expectations *efficiently* requires a distinct computational pipeline to that of modern PPSs.

Some PPSs also provide syntactic sugars for forming expectation estimates from the samples produced by inference, but these do not adjust the inference itself to exploit target function information. For example, in Stan [Carpenter et al., 2017] users can apply target functions to posterior samples using the `generated_quantities` block. Similarly, in Pyro [Bingham et al., 2019] the return values are stored along with MCMC posterior samples, thus allowing expectations to be estimated by taking empirical averages. PyMC3 [Salvatier et al., 2016] allows users to track deterministic transformations of the latent variables. Turing itself also provides a `generated_quantities` function, similar to Stan (see Appendix C for an example).

5 EXPERIMENTS

We demonstrate the effectiveness of the EPT target-aware inference methods on three problems: a synthetic numerical example, an SIR epidemiology model, and a Bayesian hierarchical model of Radon concentration. Our EPT implementation and the code for all experiments is contained in the supplementary material.

The performance of EPT depends on the performance of the chosen marginal likelihood estimator. At the time of writing, Turing provides implementations of Sequential Monte Carlo [Del Moral et al., 2006] and Importance Sampling (IS) as inference algorithms that provide marginal likelihood estimates, but only allows using the prior as the proposal which can never be target-aware. To address this issue, we implemented a new Turing inference engine that uses Annealed Importance Sampling (AnIS) [Neal, 2001] (see Appendix A). We chose AnIS because it is effective at estimating the normalization constant even for high-dimensional models [Wallach et al., 2009, Salakhutdinov and Larochelle, 2010, Wu et al., 2017].

AnIS requires setting two hyperparameters: an annealing schedule and a transition kernel. Currently, users can choose between two transition kernels: Metropolis-Hastings (MH) implemented in `AdvancedMH.jl` [Turing Development Team, 2020] and Hamiltonian Monte Carlo (HMC) [Neal, 2011, Hoffman and Gelman, 2014, Betancourt, 2018] in `AdvancedHMC.jl` [Xu et al., 2020]. To ensure a fair comparison we use the same setup and hyperparameters for both EPT's backend and standard, non-target-aware AnIS. We also compare directly to MCMC targeting the posterior and using the same type of transition kernel as AnIS and EPT. This transition kernel is MH in Section 5.1 and HMC elsewhere. Detailed configurations are given in Appendix F.

To compare the performance of the estimators we look at the effective sample size (ESS) and the relative squared error (RSE) $\hat{\delta} := (\hat{\mu} - \mu)^2/\mu^2$, where μ denotes the ground-truth value and $\hat{\mu}$ is the estimate. All our experiments correspond to target functions which are always positive, so we use Z_1 to refer to Z_1^+ as $Z_1^- = 0$. Appendix E shows how EPT can avoid computation for Z_1^- when possible.

Both EPT's backend and AnIS produce unnormalized weighted samples w_i so the ESS is calculated as $(\sum_i w_i)^2/\sum_i w_i^2$. For MCMC we focus solely on the RSE, as MCMC ESS estimates, which are based on autocorrelation [Vehtari et al., 2020], are not directly comparable with ESS estimates based on importance weights, and unreliable if the RSE is large.

For EPT 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. We use $\text{ESS}_{\text{AnIS retargeted}}$ to denote what the ESS would be if we use the AnIS samples to estimate Z_1 by multiplying the weights generated by AnIS with $f(x)$.

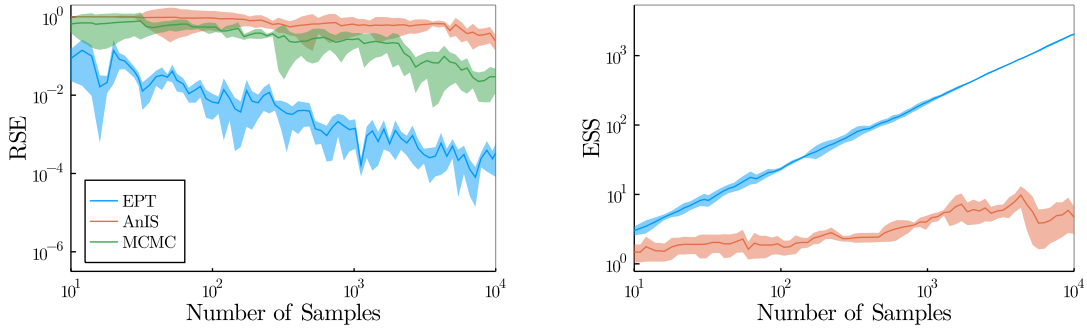


Figure 3: Relative squared error and effective sample size for the Gaussian posterior predictive experiment. The solid lines show the median of the estimator while the shaded region show the 25 % and 75 % quantiles. Medians and quantiles are computed over 10 separate runs with different random seed for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

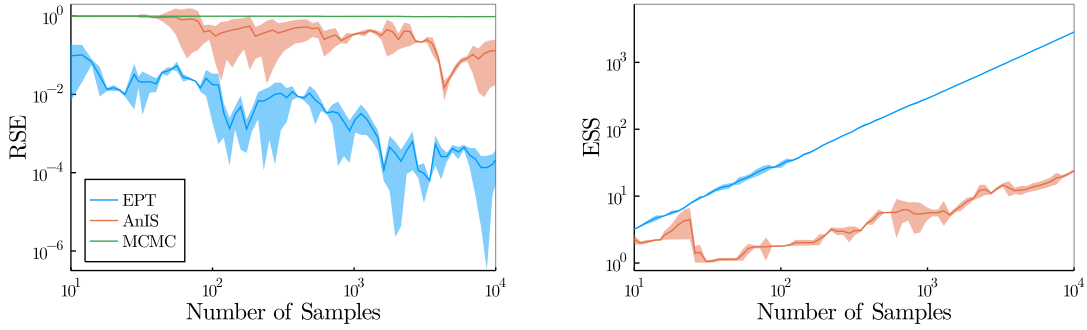


Figure 4: Relative squared error and effective sample size for the SIR experiment. Conventions as in Figure 3. Medians and quantiles are computed over 5 runs for different random seeds.

When comparing our estimators based on RSE and ESS we ensure that we give each estimator the same computational budget i.e. same number of likelihood evaluations. Thus if we have an AnIS estimator with n intermediate distributions and K samples, then we compare it to the EPT estimator which uses n intermediate distributions and $K/2$ samples for each of the two separate terms. In turn, we compare AnIS and EPT to an MCMC estimator with $n \cdot K$ samples.

5.1 GAUSSIAN POSTERIOR PREDICTIVE

The first problem considered is calculating the posterior predictive distribution of a Gaussian model with an unknown mean, where $\gamma(\mathbf{x}) = \mathcal{N}(\mathbf{x}; 0, I)\mathcal{N}(\mathbf{y}; \mathbf{x}, I)$ and $f(\mathbf{x}) = \mathcal{N}(-\mathbf{y}; \mathbf{x}, \frac{1}{2}I)$ are the unnormalised density and target function, respectively. We assume our observed data is $\mathbf{y} = (3.5/\sqrt{10})\mathbf{1}$ where $\mathbf{1}$ is a 10-dimensional vector of ones. Using EPT we can express this expectation in just 5 lines of code—the full model is given in Appendix J. This problem is amenable to an analytic solution so allows us to compute the error of the estimates. Figure 3 compares the performance of EPT, AnIS, and MCMC (here MH). We see a clear benefit to using the target-aware inference algorithm to estimate the expectation. EPT achieves a lower RSE, and the ESS highlights the advantage of using separate marginal likelihood estimators for Z_1 and Z_2.

5.2 SIR EPIDEMIOLOGICAL MODEL

Our second problem setting is a more applied example based on the Susceptible-Infected-Recovered (SIR) model of Kermack et al. [1927] from the field of epidemiology. Assume we face a disease outbreak. The government has provided us with a function yielding the expected cost of the disease which depends on the basic reproduction rate R_0, which indicates the expected number of people one infected person will infect in a population where everyone is susceptible. We seek to infer R_0 and the expected cost of the outbreak.

The SIR model divides the population into three compartments: people who are susceptible to the disease, those who are currently infected, and those who have already recovered. The dynamics of the outbreak are modelled by a set of differential equations

$$\frac{dS}{dt} = -\beta S \frac{I}{N}, \quad \frac{dI}{dt} = \beta S \frac{I}{N} - \gamma I, \quad \frac{dR}{dt} = \gamma I, \quad (4)$$

with parameters β and γ. S, I and R correspond to the number of people susceptible, infected and recovered, respectively. The size of the total population is $N = S+I+R$. 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 $R_0 = \beta/\gamma$. We assume γ to be

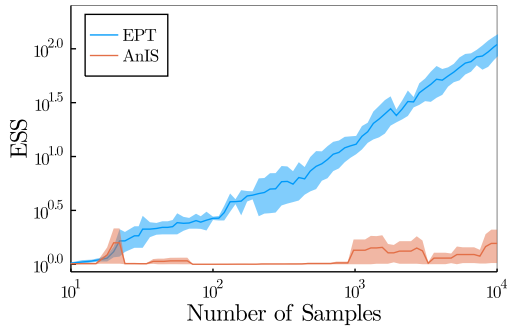


Figure 5: ESS plots for the Radon experiment. Conventions as in Figure 3; estimates based on 10 runs/seeds.

known, and we want to infer β and the initial number of infected people I_0. The full statistical model along with the cost function which is defined with respect to R_0 is given in Appendix G.

This scenario is a good use case for EPT because we are interested in estimating a specific expectation with high accuracy. Furthermore, our cost function has some outcomes which might have low probability under the posterior but which incur a very high cost. These outcomes are liable to be missed by non-target-aware schemes, leading to extremely skew estimators that almost always underestimate the expectation.

Figure 4 compares the performance of the estimators. Since this problem is not amenable to an analytic solution, we estimate the ground-truth using a customized IS estimator with orders of magnitude more samples than estimates presented in the plot (see Appendix F). We see that our approach substantially improves on the baselines, with MCMC (here HMC) failing to provide any meaningful estimate; it produces no samples where $f(x)$ is significant. EPT is able to overcome this through its use of a separate estimator for $\gamma(x)f(x)$. The fact that MCMC does far worse than AnIS, despite neither being target-aware, stems from the latter producing a greater diversity of (weighted) samples, a small number of which land in regions of high $f(x)$ by chance, see Appendix L for discussion.

5.3 HIERARCHICAL RADON CONCENTRATION MODEL

Our third problem setting is a Bayesian hierarchical model for the radon concentration in households in different counties, adapted from Gelman and Hill [2006]. For the jth house in county i, we would like to predict the log radon concentration y_{ij} inside the house. For each house we have a covariate x_{ij} which is 0 if the house has a basement, and 1 if it does not. With this setup, the model is defined as

$$\mu_\alpha \sim \mathcal{N}(0, 10), \qquad \alpha_i \sim \mathcal{N}(\mu_\alpha, 0.12), \tag{5}$$

$$\mu_\beta \sim \mathcal{N}(0, 10), \qquad \beta_i \sim \mathcal{N}(\mu_\beta, 0.22), \tag{6}$$

$$\epsilon \sim \text{HalfCauchy}(0, 5), \qquad y_{ij} \sim \mathcal{N}(\alpha_i + \beta_i x_{ij}, \epsilon). \tag{7}$$

Table 1: Final estimates for the Radon experiments. Mean and standard deviation estimated over 10 runs.

METHOD	FINAL ESTIMATE
EPT	3.74e−8 ± 2.39e−9
ANIS	1.15e−9 ± 3.02e−9
MCMC	7.79e−18 ± 2.46e−17

We now want to find out whether the radon level in *all* households is below an acceptable level, taking this threshold to be 4pCi/L. The probability of this event is equal to the expectation under the posterior of a step function $f(x)$. However, to allow the use of HMC transition kernels we use a logistic function as a continuous relaxation of this step function. See Appendix H for more details.

This problem cannot be solved analytically and estimating the ground-truth with sufficient accuracy is computationally infeasible. We, therefore, resort to comparing EPT and AnIS based on their ESSs, noting that a low ESS almost exclusively means a poor inference estimate, while a high ESS is a strong (but not absolute) indicator of good performance. As we can see in Figure 5, EPT outperforms standard AnIS by several orders of magnitude. Additionally, Table 1 presents the final expectation estimates for each method. All methods differ in their estimates. However, EPT is the only one where the standard deviation of the estimate is small relative to its mean estimate, which, coupled with our ESS results, provides strong evidence that it is significantly outperforming the baselines. In particular, it seems clear that the MCMC (here HMC) estimate is very poor, as its estimate is many orders of magnitude smaller than the others, confirming that it has failed to produce any samples in regions where $f(x)$ is non-negligible.

6 CONCLUSION

We have introduced the concept of expectation programming which describes the process of encoding expectations programmatically and automating their estimation in an efficient, *target-aware* manner. This concept is realized in EPT, which is based on Turing, and we have shown that EPT estimates expectations effectively in practice through a combination of program transformations and target-aware estimators. The modularity of EPT means that it can easily be built on by others through alternative base inference algorithms or target-aware strategies.

Future work. We believe the introduction of the expectation programming concept can pave the way for exciting future advances. EPT focuses on the automation of TABI estimators but other implementations focusing on different approaches are conceivable, for example, systems targeting the automatic synthesis of control variates for a given input program, just as there are different PPSs focusing on distinct inference algorithms.

References

Michael Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo, 2018.

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.

Ming-Hui Chen, Qi-Man Shao, et al. On Monte Carlo Methods for Estimating Ratios of Normalizing Constants. *The Annals of Statistics*, 25(4):1563–1594, 1997.

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.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo Samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68 (3):411–436, 2006.

Michael Evans, Tim Swartz, et al. Methods for Approximating Integrals in Statistics with Special Emphasis on Bayesian Integration Problems. *Statistical science*, 10(3): 254–272, 1995.

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

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*, PLDI 2020, page 883–897, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386006. URL https://doi.org/10.1145/3385412.3386006.

Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press, 2006. doi: 10.1017/CBO9780511790942.

Andrew Gelman and Xiao-Li Meng. Simulating Normalizing Constants: From Importance Sampling to Bridge Sampling to Path Sampling. *Statistical science*, pages 163–185, 1998.

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

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2021-5-21.

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.

Timothy Classen Hesterberg. *Advances in Importance Sampling*. PhD thesis, Stanford University, 1988.

M. Hoffman and A. Gelman. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623, 2014.

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.

Simon Lacoste-Julien, Ferenc Huszár, and Zoubin Ghahramani. Approximate Inference for the Loss-Calibrated Bayesian. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 416–424, 2011.

Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, abs/1404.0099, 2014. URL http://arxiv.org/abs/1404.0099.

Xiao-Li Meng and Wing Hung Wong. Simulating Ratios of Normalizing Constants via a Simple Identity: a Theoretical Exploration. *Statistica Sinica*, pages 831–860, 1996.

Tom Minka, John M. Winn, John P. Guiver, Yordan Zaykov, Dany Fabian, and John Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 2018. URL https://www.sciencedirect.com/science/article/pii/S1367578818301202.

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.

Man-Suk Oh and James O Berger. Adaptive Importance Sampling in Monte Carlo Integration. *Journal of Statistical Computation and Simulation*, 41(3-4):143–168, 1992.

Art B. Owen. *Monte Carlo Theory, Methods and Examples*. 2013.

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.

Ruslan Salakhutdinov and Hugo Larochelle. Efficient Learning of Deep Boltzmann Machines. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 693–700, 2010.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic Programming in Python Using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

G.M. Torrie and J.P. Valleau. Nonphysical Sampling Distributions in Monte Carlo Free-Energy Estimation: Umbrella Sampling. *Journal of Computational Physics*, 23(2):187 – 199, 1977. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-9991(77)90121-8. URL http://www.sciencedirect.com/science/article/pii/0021999177901218.

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.

Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner. Rank-Normalization, Folding, and Localization: An Improved $\widehat{R}$ for Assessing Convergence of MCMC. *Bayesian Analysis*, 2020. doi: 10.1214/20-BA1221. URL https://doi.org/10.1214/20-BA1221. Advance publication.

Hanna M. Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. *Evaluation Methods for Topic Models*, page 1105–1112. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605585161. URL https://doi.org/10.1145/1553374.1553515.

Robert L Wolpert. Monte Carlo Integration in Bayesian Statistical Analysis. *Contemporary Mathematics*, 115: 101–116, 1991.

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

Yuhuai Wu, Yuri Burda, R. Salakhutdinov, and Roger B. Grosse. On the Quantitative Analysis of Decoder-Based Generative Models. *International Conference on Learning Representations (ICLR)*, 2017.

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 (AABI)*, 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.

Appendices for Expectation Programming: Adapting Probabilistic Programming Systems to Estimate Expectations Efficiently

Anonymous Authors

A ANNEALED IMPORTANCE SAMPLING

Annealed importance sampling (AnIS) [Neal, 2001] is an inference algorithm which was developed with the goal of efficiently estimating the normalization constant Z of an unnormalized density $\gamma(x)$. It works by defining a sequence of annealing distributions $\pi_0(x), \ldots, \pi_n(x)$ which interpolate between a simple base distribution $\pi_0(x)$ (typically the prior for a Bayesian model) and the complex target density $\pi_n(x) = \gamma(x)$. The most common scheme is to take

$$\pi_i(x) \propto \lambda_i(x) = \pi_0(x)^{1-\beta_n}\gamma(x)^{\beta_n}, \tag{8}$$

with $0 = \beta_0 < \ldots < \beta_n = 1$. The algorithm further requires the definition of Markov chain transition kernels $\tau_1(x, x'), \ldots, \tau_{n-1}(x, x')$ and proceed to generate the jth weighted sample as follows First, sample initial particle $x_j^{(1)} \sim \pi_0(x)$, then for $i = 1, \ldots, (n-1)$, generate $x_j^{(i+1)} \sim \tau_i(x_j^{(i)}, \cdot)$ and, finally, return sample $x_j^{(n)}$ with weight

$$w_j = \frac{\lambda_1(x_j^{(1)})\lambda_2(x_j^{(2)})\ldots\lambda_n(x_j^{(n)})}{\pi_0(x_j^{(1)})\lambda_1(x_j^{(2)})\ldots\lambda_{n-1}(x_j^{(n)})} \tag{9}$$

We can estimate expectations with the weights and samples just as in importance sampling. Thus we can estimate the expectation and the normalization constant as

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

B THEORETICAL DETAILS

B.1 ASSUMPTIONS IN DEFINITION 4

To ensure correctness most PPSs assume that a particular inference algorithm will converge to the distribution of F (i.e. the distribution over return values). A standard PPS Monte Carlo inference engine will now produce a sequence of samples $F_n,\ n = 1, 2, \ldots$ and consistency requires that F_n converges in distribution to F as $n \to \infty$. This is equivalent to requiring that for *any* integrable function h, $\mathbb{E}[h(F_n)] \to \mathbb{E}[h(F)]$; and it presupposes that the distribution of F is a finite measure, i.e., $\mathbb{E}[F]$ is finite. We thus see our assumption is strictly weaker than that of standard PPSs that allow return values from programs: we only need convergence in the case where h is the identity mapping, not all integrable functions.

B.2 PROOF FOR THEOREM 1

Theorem 1. *Let $\mathcal{E}$ be a valid expectation program in EPT with unnormalized density $\gamma(x_{1:n})$, defined on possible traces $x_{1:n} \in \mathcal{X}$, with return value $F = f(x_{1:n})$. Then $\gamma_1^+(x_{1:n}) := \gamma(x_{1:n})\max(0, f(x_{1:n}))$, $\gamma_1^-(x_{1:n}) := -\gamma(x_{1:n})\min(0, f(x_{1:n}))$, and $\gamma_2(x_{1:n}) := \gamma(x_{1:n})$ are all valid unnormalized probabilistic program densities. Further, if $\{\hat{Z}_1^+\}_m$, $\{\hat{Z}_1^-\}_m$, $\{\hat{Z}_2\}_m$ are sequences of estimators for $m \in \mathbb{N}^+$ such that*

$$\{\hat{Z}_1^\pm\}_m \xrightarrow{p} \int_{\mathcal{X}} \gamma_1^\pm(x_{1:n})d\mu(x_{1:n}),$$

$$\{\hat{Z}_2\}_m \xrightarrow{p} \int_{\mathcal{X}} \gamma_2(x_{1:n})d\mu(x_{1:n})$$

where $\xrightarrow{p}$ means convergence in probability as $m \to \infty$, then $(\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m)/\{\hat{Z}_2\}_m \xrightarrow{p} \mathbb{E}[F]$.

Proof. We start by noting that as $\gamma_2(x_{1:n})$ is identical to $\gamma(x_{1:n})$, it is by assumption a valid unnormalized program density. Meanwhile, by construction, $\gamma(x_{1:n})_1^+, \gamma(x_{1:n})_1^- \geq 0, \forall x_{1:n} \in \mathcal{X}$. Further, each can be written in the form of (1) by taking the correspond definition of $\gamma(x_{1:n})$ and adding in factors $\exp(\psi_{K+1}) = \max(0, f(x_{1:n}))$ and $\exp(\psi_{K+1}) = -\min(0, f(x_{1:n}))$ for $\gamma(x_{1:n})_1^+$ and $\gamma(x_{1:n})_1^-$ respectively. To finish the proof that $\gamma^{\pm}(x_{1:n})$ are valid densities, we show that $0 < Z_1^{\pm} < \infty$.

Starting with the standard definition of an expectation for arbitrary random variables, we can express $\mathbb{E}[F]$ as

$$\int_{\mathcal{X}} f(x_{1:n}) d\mathbb{P}(x_{1:n}) = \int_{\mathcal{X}} f^+(x_{1:n}) d\mathbb{P}(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) d\mathbb{P}(x_{1:n}). \tag{10}$$

Noting that if F is integrable then by the definition of the Lebesgue integral $\int_{\mathcal{X}} f^+(x_{1:n}) d\mathbb{P}(x_{1:n}) < \infty$ and $\int_{\mathcal{X}} f^-(x_{1:n}) d\mathbb{P}(x_{1:n}) < \infty$. Now inserting the distribution the program defines over $x_{1:n}$,

$$= \int_{\mathcal{X}} f^+(x_{1:n}) \pi(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) \pi(x_{1:n}) d\mu(x_{1:n}) \tag{11}$$

and noting that $\gamma(x_{1:n}) \geq 0$ for all $x_{1:n} \in \mathcal{X}$ and $0 < \int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n}) < \infty$,

$$= \frac{\int_{\mathcal{X}} f^+(x_{1:n}) \gamma(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) \gamma(x_{1:n}) d\mu(x_{1:n})}{\int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n})} \tag{12}$$

$$= \frac{\int_{\mathcal{X}} \gamma_1^+(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} \gamma_1^-(x_{1:n}) d\mu(x_{1:n})}{\int_{\mathcal{X}} \gamma_2(x_{1:n}) d\mu(x_{1:n})} =: \frac{Z_1^+ - Z_1^-}{Z_2}. \tag{13}$$

In our theorem statement we have assumed that $\{\hat{Z}_1^+\}_m \xrightarrow{p} Z_1^+$, $\{\hat{Z}_1^-\}_m \xrightarrow{p} Z_1^-$, and $\{\hat{Z}_2\}_m \xrightarrow{p} Z_2$, from which it now follows by Slutsky's Theorem that

$$\frac{\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m}{\{\hat{Z}_2\}_m} \xrightarrow{p} \frac{Z_1^+ - Z_1^-}{Z_2} = \mathbb{E}[F] \tag{14}$$

as required. □

B.3 DETAILS ABOUT EQUATION (1)

Any probabilistic program defines a 'density' function in the form of Equation (1). This definition makes sense for a large class of programs, permitting branching on random variables, higher-order functions, recursion, stochastic memoization, and conditioning on internally sampled variables [Rainforth, 2017, §4.3]. However, for this function to correspond to a valid unnormalized probability density we need to assume that a) the program halts with probability 1 and b) that the integral over the entire domain of γ with respect to the implicitly defined reference measure is finite, i.e. $Z = \int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n}) < \infty$ where μ is the reference measure and $\mathcal{X}$ denotes the space of valid program traces.

We further need to clarify our usage of the term 'density function.' In general, probabilistic programs denote measures (or kernels if there are free variables) [Kozen, 1979, Staton et al., 2016, Borgström et al., 2011]. When we talk about the density function of a probabilistic program, formally we are referring to the Radon-Nikodym derivative of the measure denoted by this program with respect to an appropriate reference measure, where this reference measure is itself implicitly defined by the program.

C ESTIMATING EXPECTATIONS IN TURING

C.1 STANDARD APPROACH

```
@model function model(y=2)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
end
```

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

f(x) = x^3
posterior_x = Array(posterior_samples[:x])
expectation_estimate = mean(map(f, 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.

C.2 USING GENERATED QUANTITIES FUNCTION

When we designed the API Turing largely ignored the `return` statements in the model definition. In the meantime Turing introduced a convenience function `generated_quantities`. Given a model and N samples it returns a list of the N return values generated by running the program on each sample. Note that `generated_quantities` reruns the entire `model` function for each posterior sample to compute the return value. This means that for models which have an expensive likelihood computation the use of `generated_quantities` might incur a significant overhead.

It is important to note that `generated_quantities` is merely a convenience function and does not change how Turing interprets model definitions. In fact, the `generated_quantities` function provides complimentary functionality and Turing models generated with EPT can use this function without problems.

The example from Section C.1 can be rewritten to use `generated_quantities`:

```
@model function model(y=2)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end

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

expectation_estimate = mean(generated_quantities(model(), posterior_samples))
```

D FULL EXAMPLE OF MACRO TRANSFORMATION

The expectation

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end
```

gets transformed into

```
@model function gamma1_plus(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    if _context isa Turing.DefaultContext
        @addlogprob!(log(max(tmp, 0)))
    end
    return tmp
end
```

```
@model function gamma1_minus(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    if _context isa Turing.DefaultContext
        @addlogprob!(log(-min(tmp, 0)))
    end
    return tmp
end

@model function gamma2(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end

expt_prog = 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.

For `gamma1_plus` and `gamma1_minus` we also have to check in what `_context` the model is executed in. Turing allows to execute the model with different contexts which change the model behaviour. For example, there is a `PriorContext` which essentially ignores the tilde statements which have observed data on the LHS. This is useful for evaluating the prior probability of some parameters. However, by default the `@addlogprob` macro ignores the model context. As a consequence if a Turing model includes an `@addlogprob` macro and is executed with a `PriorContext` then it no longer calculates the log prior probability but instead the log prior probability plus whatever value was added with the `@addlogprob` statement. Since we want to use the Turing model with Annealed Importance Sampling we need to be able to extract the prior from our model and hence we need to ensure that we do not call `@addlogprob` when executed in a `PriorContext`. This is what the added **`if`** clause ensures.

E DIFFERENT ESTIMATORS FOR Z_1^+, Z_1^- AND Z_2

The target function $f(x) = x^2$ in the following expectation is always positive:

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^2
end
```

Therefore, we already know that $Z_1^- = 0$, so it would be wasteful to spend computational resources on estimating Z_1^-. EPT allows users to specify the marginal likelihood estimator for each of the terms in TABI separately which means if the user knows that the target function is always positive they can specify that 0 samples should be used to estimate Z_1^-:

```
expct_estimate, diagnostics = estimate_expectation(
    expt_prog(2), TABI(
        TuringAlgorithm(AnIS(), num_samples=1000),  # Z_1^+
        TuringAlgorithm(AnIS(), num_samples=0),     # Z_1^-
        TuringAlgorithm(AnIS(), num_samples=1000)   # Z_2
))
```

It is easy to see how this can be adapted to the case in which we have $Z_1^+ = 0$. This interface is not just useful for avoiding unnecessary computation, in some cases the user might also want to have different marginal likelihood estimators for each term. This allows user to further tailor the inference algorithm for the given target function $f(x)$.

F HYPERPARAMETERS FOR EXPERIMENTS

EPT runs standard annealed importance sampling twice: one time to estimate Z_1^+ and the other time to estimate Z_2. For each of the problems we always use the same hyperparameters for the annealed importance sampling algorithm both to run AnIS and for the two estimates in EPT. Running times for the individual scripts are given in the code supplementary.

F.1 POSTERIOR PREDICTIVE

For the annealed importance sampling, we use a MH transition kernel with an isotropic Gaussian with covariance $0.5I$ as a proposal and 5 MH steps on each annealing distribution. We use 100 uniformly spaced annealing distributions. For the MCMC, we collect $5 \cdot 10^6$ samples in total. To parallelise sampling we run 500 chains with 10^4 samples each in parallel, discarding the first 10^3 samples as burn-in. We use a MH transition kernel with standard Normal proposal.

F.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.

For the MCMC model we collect 10^6 samples in total with Turing's implementation of NUTS and a target acceptance rate of 65%.[1] We parallelise sampling over 10^2 chains with 10^4 samples and discard the first 10^3 samples as burn-in.

The ground truth is computed using importance sampling with 10^8 samples and the prior as a proposal distribution. See Equation (15) for the full SIR model including the priors. The observed data was generated from the model described in (15) with $\beta = 0.25$, $I_0 = 100$, $N = 10^4$ and $\phi = 10$ as the overdispersion parameter of the SIR model. We generate data for 15 time steps.

F.3 RADON MODEL

We run EPT and AnIS with 200 intermediate distributions and one step of the dynamic HMC transition kernel [Betancourt, 2018, Hoffman and Gelman, 2014] on each intermediate distribution with a step size of 0.044. The step size was informed by running adaptive MCMC on the target distribution.

G SIR EXPERIMENT

We assume we are given data in the form of observations y_i, the number of observed newly infected people on day i. Fixing $\gamma = 0.25$, this gives us the statistical model

$$\beta \sim \text{TruncatedNormal}(2, 1.5^2, [0, \infty]), \qquad I_0 \sim \text{TruncatedNormal}(100, 100^2, [0, 10000]), \tag{15a}$$

$$S_0 = 10000 - I_0, \qquad R_0 = 0, \tag{15b}$$

$$\mathbf{x} = \texttt{ODESolve}(\beta, \gamma, S_0, I_0, R_0), \qquad y_i \sim \text{NegativeBinomial}(\mu = x_i, \phi = 0.5). \tag{15c}$$

Here `ODESolve` indicates a call to a numerical ODE solver which solves the set of equations (4). It outputs x_i, the predicted number of newly infected people on day i. We assume the observation process is noisy and model it using a negative binomial distribution, which is parametrised by a mean μ and an overdispersion coefficient ϕ. 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 are further given a cost function in terms of R_0, $\text{cost}(R_0) = 10^{12} * \text{logistic}(10R_0 - 30)$. Intuitively, the cost initially increases exponentially with R_0. However, the total cost also saturates for very large R_0 (as the entire population becomes infected).

[1] https://turing.ml/dev/docs/library/#Turing.Inference.NUTS

H HIERARCHICAL RADON MODEL

The data for this problem was taken from: https://github.com/pymc-devs/pymc-examples/blob/main/examples/data/radon.csv (the repository uses an MIT license; the data contains no personally identifiable information). The original data contains information about houses in 85 counties. In order to make estimating normalization constants more tractable we reduce the number of counties to 20.

Our target function is a function of predicted radon levels y_i for a typical house with a basement (i.e. $x_i = 0$) in county i; y_i is calculated using the predictive equation given in (7). We apply the function

$$f(y_i) = \frac{1}{1 + \exp(5(y_i - 4))}$$

to all the predicted radon levels and then take the product of all the f_i. Finally, to avoid floating point underflow we set a minimum value of $1\text{e}{-200}$.

I MULTIPLE EXPECTATIONS AND RESTRICTIONS ON $f(\cdot)$

The user is not restricted to defining only one expectation per model. By specifying multiple return values the user can specify multiple expectations. The `@exptectation` macro can recognise multiple return values and generates an expectation for each of them. The user can then estimate each expectation independently using **`estimate_expectation`**:

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x, x^2, x^3
end
y_observed = 3
expt_prog1, expr_prog2, expt_prog3 = expt_prog
expct1 = expt_prog1(y_observed)
expct1_estimate, diagnostics = estimate_expectation(
  expct1, method=TABI(marginal_likelihood_estimator=TuringAlgorithm(
    AnIS(), num_samples=1000)))
```

J POSTERIOR PREDICTIVE MODEL IN EPT

The expectation from Section 5.1 can be defined in just 5 lines of code with EPT:

```
@expectation function expt_prog(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
```

K SYNTAX DESIGN

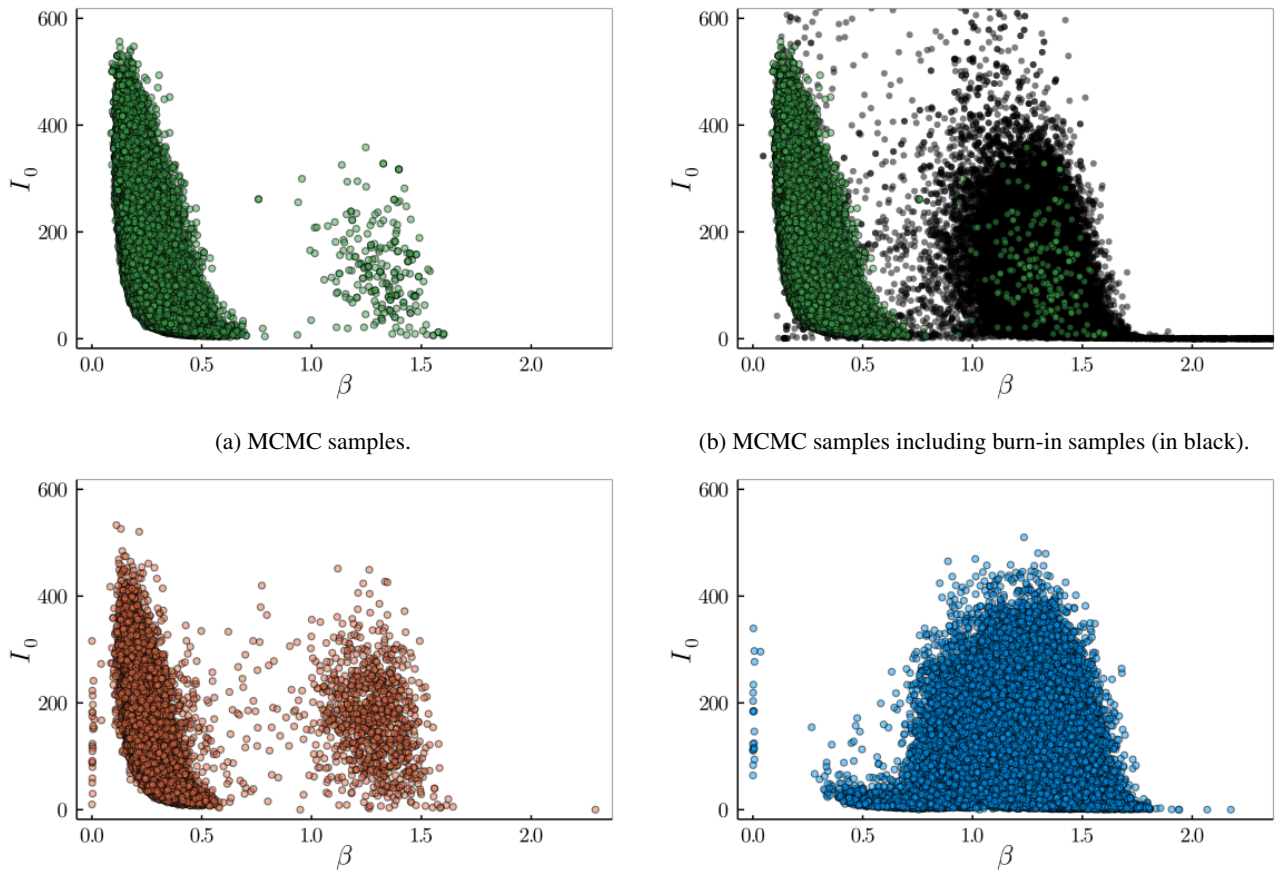
Prior works have considered two families of syntax design corresponding to the semantics required by EPT. Gordon et al. [2014] define the semantics for expectation computation via the syntax of probabilistic program's return expression, which is the approach we adopted in the design of EPT. Zinkov and Shan [2017] take a different route and define the expectation semantics via the use of syntax `expect(m, f)` where `m` is the program defining a measure and `f` is the target function.

While designing the interface of EPT we considered two different design for defining the target function: either letting users specify the target function implicitly through the return values of the function or allowing users to specify a target function `f` externally. The external function could then be passed to the **`estimate_expectation`** function explicitly.

For EPT, we decided to adopt the former of the two designs mainly due to the simplicity of the resulting user interface and implementation. In particular, it allows for simple to execute program transformations of the `@expectation` macro into

valid Turing programs to represent the individual densities, and thus the ability to use native Turing inference algorithms. Adopting the other approach would additionally require designing and specifying the interface between the function signature `f(.)` and the values of the named random draws performed by the model `m`. This would result in a more complex user-facing interface, at the slight advantage of improved compositionality of models and functions.

L SIR DISCUSSION



(a) MCMC samples.

(b) MCMC samples including burn-in samples (in black).

(c) AnIS samples.

(d) EPT samples for Z_1.

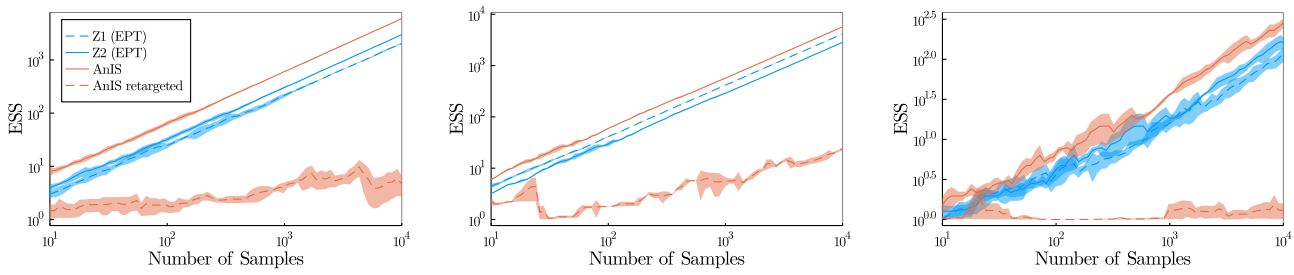
Figure 6: Samples from the different algorithms for the SIR model. Note that for Figure 6b some burn-in samples lie outside the boundaries of the plot but we adjusted the axis limits so that they are the same for all plots to allow for easier comparison.

In the SIR experiment AnIS achieved a significantly lower RSE than MCMC even though both are non-target-aware. Figure 6 shows samples from the different algorithms. The EPT samples for Z_1 visualise well in which regions of parameter space both the posterior and the target function have sufficient mass ($\beta \in [0.5, 2.0]$). The samples from AnIS and MCMC suggest that most of the posterior mass is located in the interval $\beta \in [0.3, 0.7]$. However, AnIS also generates a significant amount of samples in the parameter region $\beta \in [1.0, 1.5]$. The samples in this second "mode" are directly in the region of the target-aware samples. Further, the plots suggest that AnIS generates more samples in this regions than MCMC which is what allows AnIS to achieve a lower RSE. However, it seems that the AnIS represents the second "mode" disproportionally. Specifically looking at the burn-in samples from MCMC in Figure 6b shows that MCMC will converge to the parameter space in $\beta \in [0.3, 0.7]$ even if the initial parameter samples are around $\beta \in [1.0, 1.5]$. This indicates that this is not a failure of MCMC to detect another mode but rather that there is negligible posterior mass in that parameter region. Therefore the better performance of AnIS compared to MCMC seems to occur mostly because AnIS got lucky by accidentally generating samples in the right parameter region.

L.1 A NOTE ON MCMC ESS

The SIR experiment provides a good example of how the MCMC ESS [Vehtari et al., 2020] is unreliable for our use case. As detailed in Section F.2 for MCMC we run 100 chains with $10,000$ samples each. This is replicated 5 times to get estimates on the variability in behaviour. After discarding the burn-in samples for each chain the 5 replications give us the following final ESS estimates: $[631, 360;\ 805, 868;\ 873, 269;\ 665, 683;\ 5, 114]$. We observe that all but one replication give disproportionally high ESS estimates. We found that the replication which gives a more conservative ESS estimate of $5, 114$ is the replication which generated samples in the parameter region $\beta \in [1.0, 1.5]$ (see Figure 6a). More importantly, the MCMC ESS estimates do not seem to show any correlation with the RSE values (see Figure 4) which is the more important metric because it directly measures the error in our estimate. Therefore, we decided against using the MCMC ESS in our evaluation because it can give the impression that MCMC is performing well when it is actually failing dramatically (in terms of RSE).

M EFFECTIVE SAMPLE SIZE



(a) Gaussian Posterior Predictive. (b) SIR. (c) Radon.

Figure 7: Individual ESS values as defined in Section 5 for the three different experiments. Instead of taking $\min(\text{ESS}_{Z_1}, \text{ESS}_{Z_2})$ for EPT and $\min(\text{ESS}_{\text{AnIS retargeted}}, \text{ESS}_{\text{AnIS}})$ for AnIS we plot each value individually. Plotting each ESS value separately shows that the performance of AnIS is severely limited by its ability to generate samples in regions in which the target function $f(x)$ is large. This is indicated by the low values for $\text{ESS}_{\text{AnIS retargeted}}$.

N POSITIVE AND NEGATIVE TARGET FUNCTIONS

To demonstrate that EPT is also beneficial for target functions which are positive and negative we provide a brief description of a synthetic experiment. We assume the following model which gives us a banana shaped density (see Figure 8):

```
@expectation function banana()
    x1 ~ Normal(0, 4)
    x2 ~ Normal(0, 4)
    @addlogprob!(banana_density(x1, x2))
    return banana_f(x1, x2)
end

banana_density(x1, x2) = -0.5*(0.03*x1^2+(x2/2+0.03*(x1^2-100))^2)
```

Note that there is no observed data in this experiment which is why we chose to express the banana distribution as an unnormalized density (i.e. use the `@addlogprob!` primitive). Our target function is given by

```
function banana_f(x1, x2)
    cond = 1 / (1 + exp(50 * (x2 + 5)))
    return cond * (x1 - 2)^3
end
```

Note that the target function can be positive and negative. Figure 8 shows the RSE for EPT and AnIS. We used an MH transition kernel and 200 intermediate potentials for the Annealed Importance Sampling estimators. The RSE of AnIS does not improve because it fails to generate samples in the regions in which the target $f(x)$ is large. Rainforth et al. [2020] provide a comparison to MCMC on a similar problem so we omit it here.

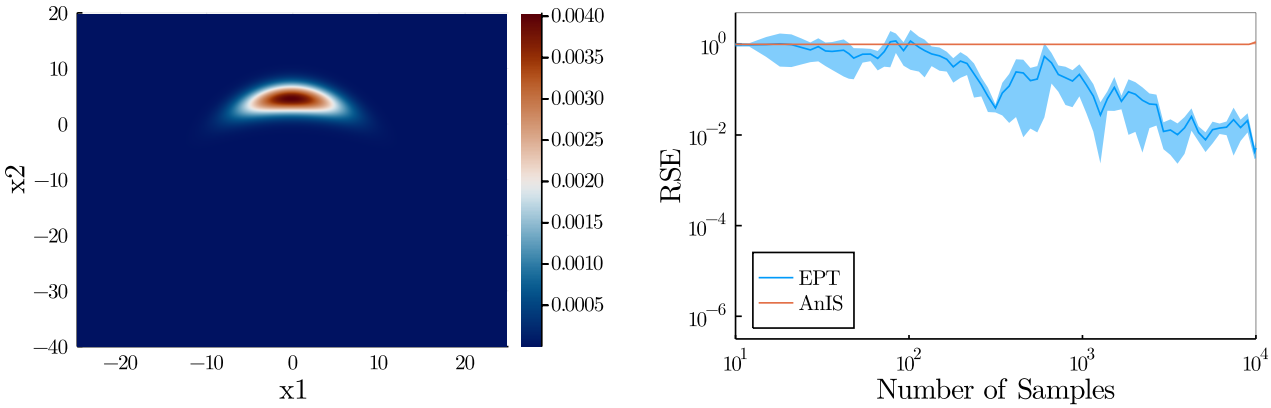


Figure 8: Banana experiment. [Left] Heatmap of the density of the model. [Right] Relative Squared Error for EPT and AnIS.

References

Michael Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo, 2018.

Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. In *European Symposium on Programming*, pages 77–96. Springer, 2011.

M. Hoffman and A. Gelman. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15:1593–1623, 2014.

Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 101–114. IEEE, 1979.

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.

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

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.

Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner. Rank-Normalization, Folding, and Localization: An Improved $\widehat{R}$ for Assessing Convergence of MCMC. *Bayesian Analysis*, 2020. doi: 10.1214/20-BA1221. URL https://doi.org/10.1214/20-BA1221. Advance publication.