Expectation Programming: Adapting Probabilistic Programming Systems to Estimate Expectations Efficiently (Supplementary Material)

Tim Reichelt[1] **Adam Goliński**[1] **Luke Ong**[1] **Tom Rainforth**[1]

[1]University of Oxford, Oxford, United Kingdom

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

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

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 j^{th} 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

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

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

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

A.1 IMPLEMENTATION DETAILS OF TURING INFERENCE ENGINE

The implementation of our new Turing inference engine is available at `https://github.com/treigerm/AnnealedIS.jl`. It is a stand-alone package that can be used completely independently from EPT and is therefore useful for any Turing user who wishes to run AnIS on their model. Furthermore, our implementation leverages the modularity of the Turing ecosystem by using existing MCMC transition kernels from the packages `AdvancedMH.jl` [Turing Development Team, 2020] and `AdvancedHMC.jl` [Xu et al., 2020].

Keeping the same notation as above, given a Turing model the AnIS inference creates Julia functions for the prior density $\pi_0(x)$ and the unnormalized density $\gamma(x)$. The unnormalized density of the program is evaluated as described in Section 2.1 and the prior density is evaluated similarly but ignores all the 'likelihood' terms $h_j(y_j \mid \phi_j)$ and all the terms added with `@addlogprob` primitive. Once we have Julia functions for $\pi_0(x)$ and $\gamma(x)$ it is straightforward to create a function for the intermediate targets $\lambda_i(x)$ for a given β_i. The Julia function for the intermediate targets $\lambda_i(x)$ can then be used by one of the MCMC samplers in `AdvancedMH.jl` or `AdvancedHMC.jl` to collect samples from the intermediate distributions.

Accepted for the 38th Conference on Uncertainty in Artificial Intelligence (UAI 2022).

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

$$\int_{\mathcal{X}} f(x_{1:n}) d\mathbb{P}(x_{1:n}) = \int_{\mathcal{X}} f^+(x_{1:n}) d\mathbb{P}(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) d\mathbb{P}(x_{1:n}). \tag{3}$$

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 $x_{1:n}$,

$$= \int_{\mathcal{X}} f^+(x_{1:n}) \pi(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) \pi(x_{1:n}) d\mu(x_{1:n}) \tag{4}$$

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

$$= \frac{\int_{\mathcal{X}} f^+(x_{1:n}) \gamma(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} f^-(x_{1:n}) \gamma(x_{1:n}) d\mu(x_{1:n})}{\int_{\mathcal{X}} \gamma(x_{1:n}) d\mu(x_{1:n})} \tag{5}$$

$$= \frac{\int_{\mathcal{X}} \gamma_1^+(x_{1:n}) d\mu(x_{1:n}) - \int_{\mathcal{X}} \gamma_1^-(x_{1:n}) d\mu(x_{1:n})}{\int_{\mathcal{X}} \gamma_2(x_{1:n}) d\mu(x_{1:n})} =: \frac{Z_1^+ - Z_1^-}{Z_2}. \tag{6}$$

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

$$\frac{\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m}{\{\hat{Z}_2\}_m} \xrightarrow{p} \frac{Z_1^+ - Z_1^-}{Z_2} = \mathbb{E}[F] \tag{7}$$

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.

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 $5 \cdot 10^7$ samples in total. To parallelise sampling we run $5 \cdot 10^3$ 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 (8) for the full SIR model including the priors. The observed data was generated from the model described in (8) 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.

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

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

$$\beta \sim \text{TruncatedNormal}(2, 1.5^2, [0, \infty]), \qquad I_0 \sim \text{TruncatedNormal}(100, 100^2, [0, 10000]), \tag{8a}$$

$$S_0 = 10000 - I_0, \qquad R_0 = 0, \tag{8b}$$

$$\mathbf{x} = \texttt{ODESolve}(\beta, \gamma, S_0, I_0, R_0), \qquad y_i \sim \text{NegativeBinomial}(\mu = x_i, \phi = 0.5). \tag{8c}$$

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 **?**.

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).

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.

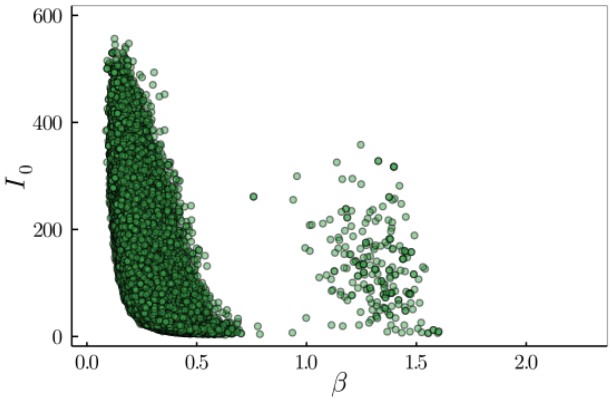
For EPT, we decided to adopt the former of the two designs mainly due to the simplicity of the resulting user interface and implementation. In particular, it allows for simple to execute program transformations of the `@expectation` macro into valid Turing programs to represent the individual densities,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

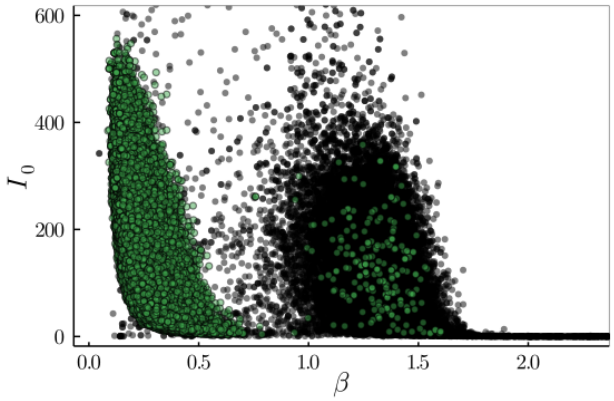
In the SIR experiment AnIS achieved a significantly lower RSE than MCMC even though both are non-target-aware. Figure 1 shows samples from the different algorithms. The EPT samples for Z_1 visualise well in which regions of parameter space both the posterior and the target function have sufficient mass ($\beta \in [0.5, 2.0]$). 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 1b 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

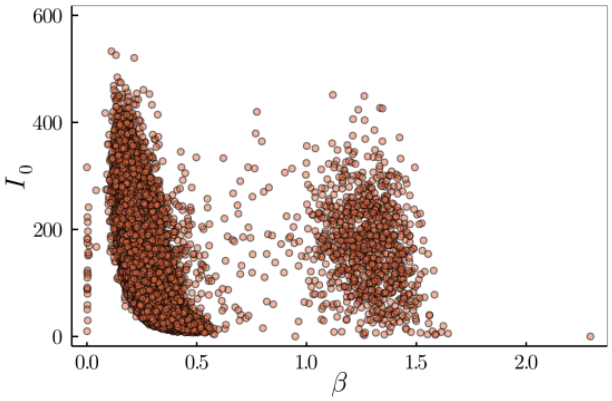
The SIR experiment provides a good example of how the MCMC ESS [Vehtari et al., 2020] is unreliable for our use case. As detailed in Section F.2 for MCMC we run 100 chains with $10,000$ samples each. This is replicated 5 times to get estimates on the variability in behaviour. After discarding the burn-in 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 1a). 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).



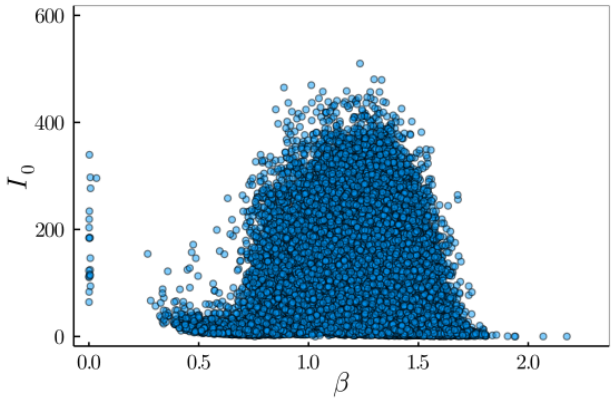
(a) MCMC samples.



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



(c) AnIS samples.



(d) EPT samples for Z_1.

Figure 1: Samples from the different algorithms for the SIR model. Note that for Figure 1b 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.

L.2 ADDITIONAL STAN MCMC BASELINE

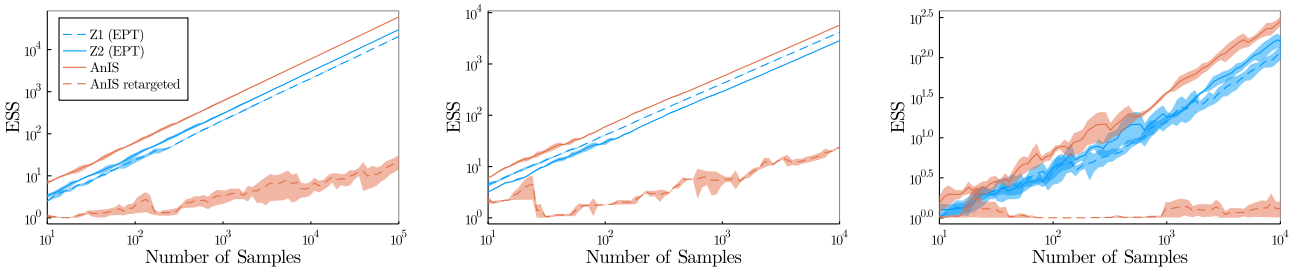
To validate our MCMC baseline we reimplemented the SIR model in Stan and used Stan's built-in default MCMC sampler. We expressed the expectation within the `generated_quantities` block leveraging the functionality described in Section 4. We have picked Stan because its built-in MCMC sampler can be reasonably considered the state-of-the-art in its domain and has been extensively tested for correctness. As shown in Table 1, Stan gives results that are similar to our current MCMC baseline (and potentially even a little worse). This demonstrates that the differences between existing PPSs are negligible compared to the effect of making inference target-aware.

Table 1: Quantiles of the RSE for different methods (the same performance metric as Figure 4, left); computed over 5 runs.

METHOD	25% QUANTILE	MEDIAN	75% QUANTILE
EPT	2.96e−6	8.10e−6	2.92e−4
ANIS	0.02	0.13	0.15
MCMC (TURING)	0.96	0.97	0.97
MCMC (STAN)	1.00	1.00	1.00

M EFFECTIVE SAMPLE SIZE

In Figure 2 we plot all the individual ESS values for EPT and the AnIS baseline. 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 $\text{ESS}_{Z_1}^{\text{AnIS}}$.



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

Figure 2: 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 $\min(\text{ESS}^{\text{AnIS}}_{Z_1}, \text{ESS}^{\text{AnIS}}_{Z_2})$ for AnIS we plot each value individually.

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 3):

```
@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 3 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.

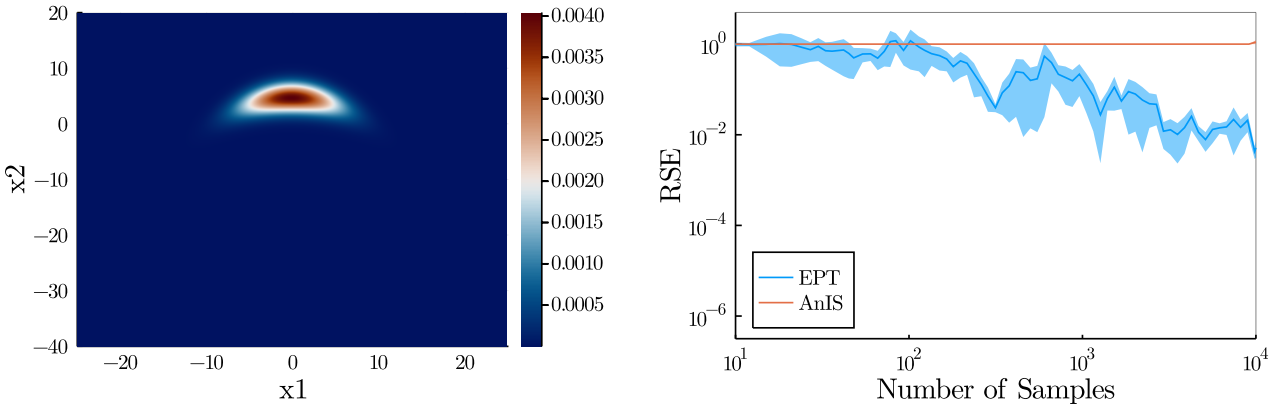


Figure 3: Banana experiment. [Left] Heatmap of the density of the model. [Right] Relative Squared Error for EPT and AnIS.

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.

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

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.

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.