# Nesting Probabilistic Programs

**Tom Rainforth**
Department of Statistics
University of Oxford
rainforth@stats.ox.ac.uk

## Abstract

We investigate the statistical implications of nesting probabilistic programming queries. We demonstrate that query nesting allows the definition of models which could not otherwise be expressed, but that changes are required to the approaches employed by most existing systems to ensure consistent estimation. We delineate possible ways one might want to nest queries and assert their respective correctness and required conditions. We further identify special cases which can be rearranged to single queries or for which conventional results apply. Our results both reaffirm the need for query nesting and highlight potential pitfalls associated with doing so.

## 1 Introduction

Probabilistic programming systems (PPSs) allow probabilistic models to be represented in the form of a generative model and statements for conditioning on data [7, 9]. Informally, one can think of the generative model as the definition of a prior, the conditioning statements as the definition of a likelihood, and the output of the program as samples from a posterior distribution. Their core philosophy is to decouple model specification and inference, the former corresponding to the user-specified program code and the latter to an inference engine capable of operating on arbitrary programs. Removing the need for users to write inference algorithms significantly reduces the burden of developing new models and makes effective statistical methods accessible to non-experts.

Some, so-called universal, systems [7, 8, 14, 29] further allow the definition of models that would be hard, or even impossible, to convey using conventional frameworks such as graphical models. One enticing manner they do this is by allowing arbitrary nesting of models, known in the probabilistic programming literature as queries [7], such that it is easy to define and run nested inference problems [7, 15, 21, 26]. This allows the definition of models that could not be encoded without nesting, such as those involving agents reasoning about other agents [13] and experimental design problems [20]. For example, one might use such nesting to model a poker player reasoning about another player as shown in Appendix B. See also [26] for a number of other examples.

Unfortunately, performing inference for nested queries falls outside the scope of conventional convergence proofs and so additional work is required to prove the validity of PPS inference engines for nested queries. Such problems constitute cases of *nested estimation*. In particular, the use of Monte Carlo (MC) methods by most PPSs mean they forms a particular instances of *nested Monte Carlo* (NMC) estimation. Recent work [5, 11, 22, 24] has demonstrated the consistency and convergence rates for NMC estimation for general classes of models. In particular, Rainforth et al. [24] provide results for cases of multiple levels of nesting as can occur for PPSs. These results show that NMC is consistent, but that it entails a convergence rate which decreases exponentially with the depth of the nesting in terms of the total computational cost. Furthermore, additional assumptions are required to achieve this convergence, most noticeably, except in a few special cases, one needs to drive not only the total number of samples used to infinity, but also the number of samples used at each layer of the estimator [24]. In other words, if an outer estimator uses $N_0$ samples of an inner estimator, each of which themselves use $N_1$ samples (giving $T = N_0 N_1$ total samples), then it is necessary for both $N_0 \to \infty$ and $N_1 \to \infty$ to achieve convergence, a requirement generally flaunted by PPSs.

The aim of this work is to establish the implications of these results on the statistical correctness of nesting probabilistic program queries. We consider different ways one might nest queries and assess their respective correctness. Namely, we consider *sampling* from the conditional distribution

of another query, *observing* another query, and using *estimates as variables* in another query. We further outline some special cases where standard MC convergence can be achieved. At times, we will *Anglican* [28, 29] syntax as a basis for discussion, an introduction for which is provided in Appendix A, noting that our results apply more generally. We find that changes are required to how Anglican tackles nested query problems to ensure consistency and allow exploitation of the special cases.

## 2 Nested Monte Carlo

We start by providing a brief introduction to NMC, referring to reader to [24] (from which we inherit our notation) for a more complete introduction. To define what we mean by a nested estimation we first assume some fixed integral depth $D \geq 0$ (with $D = 0$ corresponding to conventional estimation), and real-valued functions $f_0, \ldots, f_D$. We then recursively define $y^{(k)} \sim p\left(y^{(k)}|y^{(0:k-1)}\right)$,

$$\gamma_D\left(y^{(0:D-1)}\right) = \mathbb{E}\left[f_D\left(y^{(0:D)}\right)\middle|y^{(0:D-1)}\right] \quad \text{and}$$
$$\gamma_k(y^{(0:k-1)}) = \mathbb{E}\left[f_k\left(y^{(0:k)}, \gamma_{k+1}\left(y^{(0:k)}\right)\right)\middle|y^{(0:k-1)}\right] \quad \text{for} \quad 0 \leq k < D.$$

Our goal is to estimate $\gamma_0 = \mathbb{E}\left[f_0\left(y^{(0)}, \gamma_1\left(y^{(0)}\right)\right)\right]$, for which we use NMC:

$$I_D\left(y^{(0:D-1)}\right) = \frac{1}{N_D}\sum_{n=1}^{N_D} f_D\left(y^{(0:D-1)}, y_n^{(D)}\right) \quad \text{and}$$
$$I_k\left(y^{(0:k-1)}\right) = \frac{1}{N_k}\sum_{n=1}^{N_k} f_k\left(y^{(0:k-1)}, y_n^{(k)}, I_{k+1}\left(y^{(0:k-1)}, y_n^{(k)}\right)\right) \quad \text{for} \quad 0 \leq k < D,$$

where each $y_n^{(k)} \sim p\left(y^{(k)}|y^{(0:k-1)}\right)$ is drawn independently. Note that there are multiple values of $y_n^{(k)}$ for each possible $y^{(0:k-1)}$ and that $I_k\left(y^{(0:k-1)}\right)$ is still a random variable given $y^{(0:k-1)}$. As shown in [24], then if each $f_k$ is continuously differentiable the MSE satisfies

$$\mathbb{E}\left[(I_0 - \gamma_0)^2\right] \leq \frac{\varsigma_0^2}{N_0} + \left(\frac{C_0\varsigma_1^2}{2N_1} + \sum_{k=0}^{D-2}\left(\prod_{d=0}^{k} K_d\right)\frac{C_{k+1}\varsigma_{k+2}^2}{2N_{k+2}}\right)^2 + O(\epsilon). \tag{1}$$

where $K_k$ and $C_k$ are bounds on the magnitude of the first and second derivatives of $f_k$ respectively, $O(\epsilon)$ represents asymptotically dominated terms, and we assume

$$\varsigma_k^2 := \mathbb{E}\left[\left(f_k\left(y^{(0:k)}, \gamma_{k+1}\left(y^{(0:k)}\right)\right) - \gamma_k\left(y^{(0:k-1)}\right)\right)^2\right] < \infty \quad \forall k \in 0, \ldots, D.$$

We see that if any of the $N_k$ remain fixed, there is a minimum error that can be achieved: convergence requires each $N_k \to \infty$. For a given total sample budget $T = N_0 N_1 \ldots N_D$, the bound is tightest when $N_0 \propto N_1^2 \propto \cdots \propto N_D^2$ giving a convergence rate of $O(1/T^{\frac{2}{D+2}})$.

## 3 Nested Sampling

One of the clearest ways one might want to nest queries is by sampling from the conditional distribution of one query inside another query. A number of examples of this are provided for Church in [26]. Such problems fall under a more general framework of *nested inference* [15] or equivalently inference for so-called *doubly (or multiply) intractable* distributions [18, 26]. The key feature of these problems is that they include terms with unknown, *parameter dependent*, normalization constants. For nested probabilistic programming queries, this is manifested in *conditional normalization* within a query. Consider as an example the following Anglican model using a nested calling structure

```
(defm inner [y D]
 (let [z (sample (gamma y 1))]
  (observe (normal y z) D)
  z))
```

```
(defquery outer [D]
 (let [y (sample (beta 2 3))
       z (inner y D)]
  (* y z)))
```

compared to the following nested query model

```
(defquery inner [y D]
 (let [z (sample (gamma y 1))]
  (observe (normal z y) D)
  z))
```

```
(defquery outer [D]
 (let [y (sample (beta 2 3))
       dist (conditional inner)
       z (sample (dist y D))]
  (* y z)))
```

where we have replaced the function deceleration `defm` with a query nesting using the `conditional` special form. The unnormalized distribution on traces for the first model is straightforwardly given by

$$\tilde{\pi}_1(z, y, D) = p(y)p(z|y)p(D|y, z) = \textsc{Beta}(y; 2, 3)\textsc{Gamma}(z; y, 1)\mathcal{N}(D; z, y^2),$$

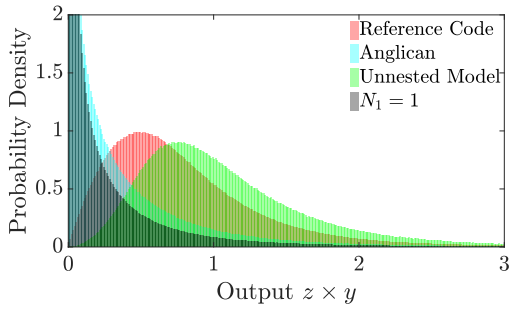



Figure 1: Histograms showing a comparison of the output samples (i.e. of $z \times y$) produced by the nested Anglican queries given in the text to approximate ground truth reference, the "unnested" model, and a naïve estimation scheme where $N_1 = 1$.

for which we can use conventional Monte Carlo inference schemes. The second model differs in that $p(z|y)p(D|y,z)$ is conditionally normalized before being used in the outer query

$$\tilde{\pi}_2(z,y,D) = p(y)p(z|y,D) = p(y)\frac{p(z|y)p(D|y,z)}{\int p(z|y)p(D|y,z)dz} = p(y)\frac{p(z|y)p(D|y,z)}{p(D|y)} \neq \tilde{\pi}_1(z,y,D).$$

The key point here is that the partial normalization constant $p(D|y)$ depends on $y$ and so $\tilde{\pi}_2(z,y,D)$ is doubly intractable because we cannot evaluate our unnormalized target distribution exactly. By normalizing $p(z|y)p(D|y,z)$ within the outer query, **conditional** has changed the distribution defined by the program. Another interesting way of looking at this is that wrapping `inner` in **conditional** has "protected" $y$ from the conditioning in `inner` (noting that $\tilde{\pi}_1(z,y,D) \propto p(y|D)p(z|y,D)$) such that the **observe** statement only affects the probability of $z$ given $y$ and not the probability of $y$. A poker-based example showing the utility of such nested inference models is given in Appendix B.

Given that this problem cannot be rearranged to a single expectation, it now obvious to ask the question: what behavior do we need for **conditional**, or query nesting through sampling more generally, in order to ensure consistency? Without access to exact sampling methods, it follows from our converge bounds (and the nonlinear mapping originating from the inversion of the normalization constant) that asymptotic bias is, in general, unavoidable for query nesting if each call of **conditional** only carries out finite computation (see also [24, Theorem 4]).[1] In other words, we need the computational budget of each **conditional** call to become arbitrarily large for convergence.

As an sanity check for this theoretical result, we conduct an empirical evaluation to check that this behavior manifests in practice. We ran Anglican's importance sampling inference engine on the simple model introduced earlier and compared this to approximate ground truth reference code (implemented in MATLAB) where each call to `inner` internally generates $N_1 = 10^3$ samples before drawing the returned $z$ from these in proportion to the weights, giving a close approximation to $p(z|y,D)$. As shown in Figure 1, the samples produced by Anglican are substantially different to the reference code, demonstrating that the outputs do not match their semantically intended distribution. For reference, we also consider the distribution induced by the "unnested" model, where **conditional** is replaced by **defm**, and a naïve estimation scheme where a sample budget of $N_1 = 1$ is used for each call to `inner`, effectively corresponding to ignoring the **observe** statement by directly returning the first draw of $z$. We see that the unnested model defines a noticeably different distribution, while the behavior of Anglican is similar, but distinct, to ignoring the **observe** statement in the inner query. Further investigation shows that the default behavior of **conditional** in a query nesting context is equivalent to the reference code but with $N_1 = 2$, inducing a substantial bias.

A special case where consistency can be maintained without requiring infinite computation for each nested call is when no random variables are input to the inner query. A condition for this is that we use a method that provides an unbiased estimate of the partition function (e.g. sequential MC (SMC)) and factor the trace weight appropriately with the product of this partition function estimate and the internal self-normalized weight of the returned sample. If this condition is maintained, we can demonstrate the validity of the approach by noting that the normalization constant of the nested query is not parameter dependent, i.e. $p(D|y) = p(D)$ using the notation from our earlier example. It can, therefore, be factored into the outer query: the nested query now defines the same distribution as if the conditional query were replaced with a standard function, e.g. **defm** in Anglican. We can thus view inference on such problems as taking a pseudo-marginal approach [1, 2], provided that all samples returned from the inner query are properly weighted [19]. The correctness of this approach

[1] Interestingly, as discussed in Appendix C, it might actually be possible to use exact sampling methods, such as rejection sampling, to avoid this bias without requiring infinite computation for each nested call. It might also be possible to use debiasing techniques to remove the particular bias that is generated, see e.g. [6, 12].

then follows from a combination of the linear $f$ and product of expectations results of [24]. However, Anglican's current use of `conditional`, for example, does not satisfy our requirement as it only returns unweighted samples. For Anglican to provide consistent estimates in such situations, one would need to change the probabilistic semantics of `conditional`, such that it both produces a sample and simultaneously factors the outer query trace weight.

Another special case is when the variables passed to the inner query can only take on, say $C$, finite possible values. As explained in [24], such problems can always be rearranged to $C$ separate estimators such that the standard Monte Carlo error rate can be achieved. For repeated nesting, this rearrangement can be recursively applied until one achieves a complete set of non-nested estimators. This special case is one which requires rearrangement or a specialist consideration by the language back-end (as done by e.g. [4, 25, 26]), with naïve implementations leading to NMC convergence rates. It emphasizes that nested problems with continuous variables are a fundamentally different problem class to those with only bounded discrete variables (or only continuous variables at the bottom layer) and we must, therefore, be wary of presuming that results from the discrete case carry over.

## 4 Nested Observation

An alternative way one might wish to nest queries is through nested observation, whereby we condition on the output of another query being a certain value. For hard-constraints such as used in Church, this is equivalent to query nesting through sampling. However, for soft constraints it can be substantially different. The distinction is the same as between `sample` and `observe` in Anglican – we are observing an outcome of a program, rather than sampling from it. As `observe` returns no outputs, the mathematical interpretation of observing the output of another query is that we are factoring the outer program trace weight by the partition function estimate of the inner query. Presuming that our partition function estimates are unbiased, this corresponds to a particular instance of the products of expectations special case given in [24]. It, therefore, follows that if the outer estimator uses importance sampling or SMC, then such an approach is consistent through proper weighting [19]. More generally, this case corresponds to a pseudo-marginal approach where the samples from the inner estimator are never returned. We can, therefore, use existing results, e.g. [1], to further demonstrate the consistency of the approach for other choices of outer estimator.

In addition to the statistical complications just discussed, there are some significant semantic complications associated with nested conditioning. Though one may wish to simply run a nested query unmodified and factor the trace probability by the partition function estimate, e.g. to construct a PMMH sampler [2] as done in [23], this is more of a means of replacing an analytic likelihood term with an unbiased estimate, than a general purpose nested observation. Instead, we might wish to estimate the likelihood of a query producing a particular output. Unfortunately this is, in general, not possible for existing universal PPSs because the density of the output variables are affected by deterministic computations in the query and it is not usually possible to carry out the required change of variables computations automatically. However, this problem is identical to that encountered in estimating the conditional marginal likelihood in [23] presenting an interesting possibility for constructing a nested observation statement – one could use their marginal program transformation to estimate the partition function with certain `sample` outputs "clamped".

## 5 Estimates as Variables

Finally, one might wish to use estimates as first class variables in another query. In our previous examples, nonlinear usage of expectations only ever manifested through inversion of the normalization constant. These methods are therefore clearly insufficient for expressing more general nested estimation problems as would be required by, for example, experimental design. An example of this is shown in Appendix D corresponding to a generic program for carrying out Bayesian experimental design [3]. The validity of this "estimates as variables" problem class is effectively equivalent to that of NMC more generally and needs to be considered on a case-by-case basis.

## 6 Conclusions

We have demonstrated that, without access to exact sampling methods, then asymptotic bias is, in general, unavoidable when sampling from nested probabilistic program queries. Consequently, we see that the computational effort expended at each level of the nesting must increase to ensure consistency, a requirement not satisfied by existing systems. On the other hand, we have shown that observing the output of one query inside another query as a likelihood term is statistically valid, but found that doing this leads to semantical problems that require further work to overcome.

## Acknowledgements

Tom Rainforth's research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) ERC grant agreement no. 617071. The majority of this work was undertaken while he was in the Department of Engineering Science, University of Oxford, and was supported by a BP industrial grant.

## References

[1] C. Andrieu and G. O. Roberts. The pseudo-marginal approach for efficient Monte Carlo computations. *The Annals of Statistics*, pages 697–725, 2009.

[2] C. Andrieu, A. Doucet, and R. Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2010.

[3] K. Chaloner and I. Verdinelli. Bayesian experimental design: A review. *Statistical Science*, 1995.

[4] R. Cornish, F. Wood, and H. Yang. Efficient exact inference in discrete Anglican programs. 2017.

[5] G. Fort, E. Gobet, and E. Moulines. MCMC design-based non-parametric regression for rare-event. application to nested risk computations. *Monte Carlo Methods Appl*, 2017.

[6] P. W. Glynn and C.-h. Rhee. Exact estimation for Markov chain equilibrium expectations. *Journal of Applied Probability*, 51(A):377–389, 2014.

[7] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *UAI*, 2008.

[8] N. D. Goodman and A. Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. 2014.

[9] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*. ACM, 2014.

[10] R. Hickey. The Clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[11] L. J. Hong and S. Juneja. Estimating the mean of a non-linear function of conditional expectation. In *Winter Simulation Conference*, 2009.

[12] P. E. Jacob, F. Lindsten, and T. B. Schön. Smoothing with couplings of conditional particle filters. *arXiv preprint arXiv:1701.02002*, 2017.

[13] T. A. Le, A. G. Baydin, and F. Wood. Nested compiled inference for hierarchical reinforcement learning. In *NIPS Workshop on Bayesian Deep Learning*, 2016.

[14] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[15] T. Mantadelis and G. Janssens. Nesting probabilistic inference. *arXiv preprint arXiv:1112.3785*, 2011.

[16] J. Møller, A. N. Pettitt, R. Reeves, and K. K. Berthelsen. An efficient Markov chain Monte Carlo method for distributions with intractable normalising constants. *Biometrika*, 93(2):451–458, 2006.

[17] I. Murray and Z. Ghahramani. Bayesian learning in undirected graphical models: approximate MCMC algorithms. In *UAI*, pages 392–399. AUAI Press, 2004.

[18] I. Murray, Z. Ghahramani, and D. J. MacKay. MCMC for doubly-intractable distributions. In *UAI*, 2006.

[19] C. A. Naesseth, F. Lindsten, and T. B. Schön. Nested sequential Monte Carlo methods. In *ICML*, 2015.

[20] L. Ouyang, M. H. Tessler, D. Ly, and N. Goodman. Practical optimal experiment design with probabilistic programs. *arXiv preprint arXiv:1608.05046*, 2016.

[21] T. Rainforth. *Automating Inference, Learning, and Design using Probabilistic Programming*. PhD thesis.

[22] T. Rainforth, R. Cornish, H. Yang, and F. Wood. On the pitfalls of nested Monte Carlo. *NIPS Workshop on Advances in Approximate Bayesian Inference*, 2016.

[23] T. Rainforth, T. A. Le, J.-W. van de Meent, M. A. Osborne, and F. Wood. Bayesian Optimization for Probabilistic Programs. In *NIPS*, pages 280–288, 2016.

[24] T. Rainforth, R. Cornish, H. Yang, A. Warrington, and F. Wood. On the opportunities and pitfalls of nesting Monte Carlo estimators. *arXiv preprint arXiv:1709.06181*, 2017.

[25] A. Stuhlmüller and N. D. Goodman. A dynamic programming algorithm for inference in recursive probabilistic programs. In *Second Statistical Relational AI workshop at UAI 2012 (StaRAI-12)*, 2012.

[26] A. Stuhlmüller and N. D. Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

[27] D. Tolpin, J.-W. van de Meent, and F. Wood. Probabilistic programming in Anglican. Springer, 2015.

[28] D. Tolpin, J.-W. van de Meent, H. Yang, and F. Wood. Design and implementation of probabilistic programming language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, page 6. ACM, 2016.

[29] F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *AISTATS*, pages 2–46, 2014.

## A The Anglican Probabilistic Programming Language

Anglican is a universal PPL integrated into *Clojure* [10], a dynamically-typed, general-purpose, functional programming language (specifically a dialect of Lisp) that runs on the Java virtual machine and uses just-in-time compilation. There are two key things to get your head around for reading and using Clojure (and Lisp languages more generally): almost everything is a function and parentheses evaluate functions. For example, to code $a + b$ in Clojure one would write `(+ a b)` where `+` is a function taking two arguments (here `a` and `b`) and the parentheses cause the function to evaluate. More generally, Clojure uses prefix notation such that the first term in a parenthesis is always a function, with the other terms the arguments. Thus `((+ a b))` would be invalid syntax as the result of `(+ a b)` is not a function so cannot be evaluated. Expressions are nested in a tree structure so to code $2(a + b)/c$ one would write `(/ (* 2 (+ a b)) c)`. One can thus think about going outwards in terms of execution order – a function first evaluates its arguments before itself. Functions in Clojure are first class (i.e. they can be variables) and can be declared either anonymously using `(fn [args] body)`, for example `(fn [a b] (/ a b))`, or using the macro `defn` to declare a named function in the namespace, for example `(defn divide [a b] (/ a b))`. Local bindings in Clojure are defined using `let` blocks which denote a number of name-value pairs for binding variables, followed by a series of evaluations to carry out. Their return value is the output of the last of these evaluations. Thus for example `(let [a 2 b 3] (print a) (+ a b))` says let $a$ be equal to 2, $b$ be equal to 3, print $a$, and then evaluate $a + b$, thus returning 5.

Anglican inherits most of the syntax of Clojure, but extends it with the key special forms `sample` and `observe` [27, 28], between which the distribution of the query is defined. Informally, `sample` specifies terms in the prior and `observe` terms in the likelihood. More precisely, `sample` is used to make random draws $x_t \sim f_t(x_t|\Xi_t)$, where $\Xi_t$ is a subset of the variables in scope at the point of sampling, and `observe` is used to condition on data $g_s(y_s|\Lambda_s)$ with $\Lambda_s$ defined in the same way as $\Xi_t$. The syntax of `sample` is to take a *distribution object* as its only input, while `observe` takes a distribution object and an observation as input. Each distribution object further contains a sampling procedure and a density function. Anglican provides a number of *elementary random procedures* in the form of distribution classes for common sampling distributions such as the normal and Poisson distributions, but also allows users to define their own distribution classes using the `defdist` macro. Distribution objects are generated by calling the class constructor with the required parameters, e.g. `(normal 0 1)` for the unit Gaussian. Despite using predominantly the same syntax, Anglican has different *probabilistic* semantics to Clojure, i.e. code is written in the same way, but is interpreted differently.

Anglican queries are written using the macro `defquery`. This allows users to define a model using a mixture of `sample` and `observe` statements and deterministic code, and bind that model to a variable. A simple example of an Anglican query is shown in Figure 2, corresponding to a model where we are trying to infer the mean and standard deviation of a Gaussian given some data. The syntax of `defquery` is `(defquery name [args] body)` so in Figure 2 our query is named `my-query` and takes in arguments `data`. The query starts by sampling `mu`$\sim \mathcal{N}(0, 1)$ and `sig`$\sim$ GAMMA$(2, 2)$ and constructing a distribution object `lik` to use for the observations. It then maps over each datapoint and observes it under the distribution `lik`, remembering that `fn` defines a function and `map` applies a function to every element of a list or vector. Note that `observe` simply returns `nil` and so we are not interested in these outputs – they affect the probability of a program execution trace, but not the calculation of a trace itself. After the observations are made, `mu` and `sig` are returned from the `let` block and then by proxy the query itself. This particular query defines a correctly normalized joint distribution (note some queries are not normalized) given by

```
(defquery my-query [data]
 (let [mu (sample (normal 0 1))
       sig (sample (gamma 2 2))
       lik (normal mu sig)]
  (map (fn [obs]
         (observe lik obs))
    data)
  [mu sig]))
```

Figure 2: A simple Anglican query.

$$p(\mu, \sigma, y_{1:S}) = \mathcal{N}(\mu; 0, 1)\ \text{GAMMA}(\sigma; 2, 2) \prod_{s=1}^{S} \mathcal{N}(y_s; \mu, \sigma) \tag{2}$$

where we have defined $\mu :=$`mu`, $\sigma :=$`sig`, and $y_{1:S} :=$`data`.

Inference on a query is performed using the macro `doquery`, which produces a lazy infinite sequence of approximate samples from the conditional distribution and, for appropri-

ate inference algorithms, an estimate of the partition function. The syntax of `doquery` is `(doquery inf-alg model inputs & options)` where `inf-alg` specifies an inference algorithm, `model` is our query, `inputs` is a vector of inputs to the query, the & represents that there might be multiple additional inputs, and `options` is a series of option-value pairs for the inference engine. For example, to produce 100 samples from our `my-query` model with data `[2.1 5.2 1.1]` using the LMH inference algorithm with no additional options, we can write `(take 100 (doquery :lmh my-query [2.1 5.2 1.1]))`.

Although Anglican mostly inherits Clojure syntax, some complications arise from the compilation to CPS-style Clojure functions, required for performing inference. It is thus not possible to naïvely use all Clojure functions inside Anglican without providing appropriate information to the compiler to perform the required transformation. The transformation for many core Clojure functions has been coded manually, such that they form part of the Anglican syntax and can be used directly. Anglican further allows users to write functions externally to `defquery` using the macro `defm`, which is analogous to `defn` in Clojure and has the same call syntax, which can then be called from inside the query without restrictions. Other deterministic Clojure functions can also be called from inside queries but must be provided with appropriate wrapping to assist the compiler. Thankfully this is done automatically (except when the function itself is higher order) using the macro `with-primitive-procedures` that takes as inputs Clojure functions and creates an environment where these functions have been converted to their appropriate CPS forms.

For our purposes, an important element of Anglican is its ability to *nest* queries within one another. Though `doquery` is not allowed directly within a `defquery` block, one can still nest queries by using the special form `conditional` which takes a query and returns a distribution object constructor ostensibly corresponding to the conditional distribution (informally the posterior) defined by the query, for which the inputs to the query become the parameters. However, our results show that the default use of `conditional` does not have this interpretation in the nested query context. In truth, `conditional` defines a Markov chain whose samples converge in distribution to the conditional distribution defined by the query. When using `conditional` as an alternative to `doquery`, this is not problematic as estimates calculated using the outputs converge by the strong law of large numbers. However, when used to nest queries, this Markov chain is only ever run for a finite length of time (specifically one accept-reject step is carried out) and so does not produce samples from the true conditional distribution. By default, this Markov chain is generated by equalizing the output of sequence of importance samples (i.e. it is a Metropolis sampler using independent proposals). However, `conditional` does allow the specification of an inference algorithm and associated options, which in the nested query context leads to explicitly distinct distributions. Though hardly its intended use, this does actually provide a roundabout means of arbitrarily reducing the bias in Anglican's nested query inference (ignoring memory restrictions) using `(conditional inner-query :smc :number-of-particles N_k)` as each sample output by the SMC sweep convergences in distribution to the conditional distribution as $N_k \to \infty$ (noting that as a resampling step is done at the end of the sweep, all particles in a sweep have the same weight).

Though not officially supported usage, one can also nest Anglican queries by using a `doquery` in a Clojure function that is then passed to another query using `with-primitive-procedures` or using a `doquery` within a custom distribution defined by `defdist`. This allows our "estimates as variables" class of models to be encoded.

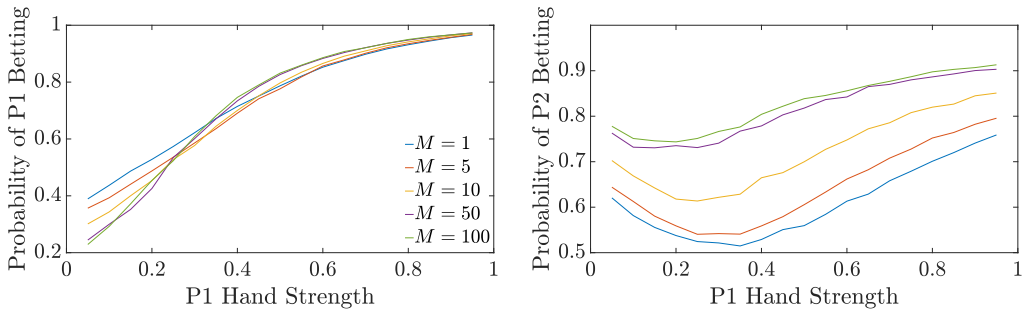
## B Poker Example

As a more realistic demonstration of the utility for allowing nested query sampling in probabilistic programs, we consider the example of simulating a poker player who reasons about another player. Anglican code for this example is given in Figure 3. In the interest of exposition, we consider the simple situation where both players are short-stacked, such that player 1's (P1's) only options are to fold or go all-in. Presuming that P1 has bet, the second player (P2) can then either fold or call. One could easily adapt the model to consider multiple players and cases where the players have more options than the binary choice of betting or not, including cases where the actions might be continuous, and where there are multiple rounds of betting (for which addition levels to the nesting would be required).

The two key functions here are the queries `p1-sim` and `p2-sim`, representing simulators for the two players. The simulations for each player exploit a utility function, which is a deterministic function of the two hands and actions taken, to reflect the preference for actions which give a bigger return. P1 acts before P2 and must thus not only reason about what hands P2 might have, but also

```
(defdist factor [] []
 ;; Factors trace probability with provided value
 (sample* [this] nil)
 (observe* [this value] value))

(defdist hand-strength []
 ;; Samples the strength of a hand
 [dist (uniform-continuous 0 1)]
 (sample* [this] (sample* dist))
 (observe* [this value] (observe* dist value)))

(defdist play-prior [hand]
 ;; A prior on whether to play a hand without reasoning about the opponent
 [dist (categorical
         {:fold (+ 0.25 (* 0.75 (- 1. hand)))
          :bet (+ 0.25 (* 0.75 hand))})]
 (sample* [this] (sample* dist))
 (observe* [this value] (observe* dist value)))

(with-primitive-procedures [hand-strength play-prior factor]
 (defm calc-payoff [player p1-hand p1-action p2-hand p2-action]
  ;; Calculate payoff given actions and hands.  Pay outs are not symmetric
  ;; due to the blinds
  (case p1-action
   :fold (if (= player 1) -1 1) ;; Pick up small blinds
   :bet (case p2-action
         :fold (if (= player 1) 2 -2) ;; Pick up big blinds
         :bet (case [player (> p2-hand p1-hand)] ;; Showdown
               [1 true] -6 [1 false] 6 [2 true] 6 [2 false] -6))))

 (defm payoff-metric [payoff]
  ;; Hueristic model converting a payoff to a log score
  (* 3 (log (/ (+ payoff 6) 12))))

 (defquery p2-sim [p2-hand p1-action]
  ;; Simulator for player 2 who know's player 1's action but not his
  ;; hand.  Returns action
  (let [p2-action (sample (play-prior p2-hand))
        p1-hand (sample (hand-strength))] ;; Simulate a hand for player 1
   (observe (play-prior p1-hand) p1-action) ;; Condition on known action
   (observe (factor) ;; Factor trace probability with payoff metric
    (payoff-metric (calc-payoff 2 p1-hand p1-action p2-hand p2-action)))
   p2-action)) ;; Return action

 (defquery p1-sim [p1-hand M]
  ;; Simulator for player 1 who reasons about what player 2
  ;; might do to choose an action given a hand
  (let [p1-action (sample (play-prior p1-hand)) ;; Sample action
        p2-hand (sample (hand-strength)) ;; Sample hand for opponent
        p2-dist (conditional p2-sim :smc :number-of-particles M)
        p2-action (if (= p1-action :fold)
                    :bet ;; No need to simulate player 2
                    (sample (p2-dist p2-hand p1-action)))]
   (observe (factor) ;; Factor trace probability with payoff metric
    (payoff-metric (calc-payoff 1 p1-hand p1-action p2-hand p2-action)))
   [p1-action p2-action])))) ;; Return actions

(defn estimate-actions [hand M]
 ;; Estimates the relative probability of betting for a hand
 (->> (doquery :importance p1-sim [hand M])
      (take 10000) collect-results empirical-distribution))
```

Figure 3: Code simulating the behavior of a poker player who reasons about the behavior of another player. Explanation provided in text.




(a) Probability player 1 betting

(b) Probability player 2 betting

Figure 4: Probabilities of each player betting output by `p1-sim` for different computational budgets of the inner estimator. The ostensibly strange behavior that P2 is more likely to bet as the strength of P1 hand's increases is because `p1-sim` conditions on having a good return for P1 and when P1 has a good hand, it becomes increasingly beneficial for that bet to be called. In other words, (b) is explicitly not the marginal distribution for optimal betting of P2 which would require a separate calculation.

the actions P2 might take. To do this, P1 utilizes the simulation for P2, which takes in as inputs an observed action from P1 and a true hand for P2, returning actions taken by P2. P2 has incomplete information – he does not know P1's hand – and so the simulation of P2's action requires inference, with a normalization constant that depends on P2's hand and P1's action, both of which are random variables in the outer query `p1-sim`. This therefore corresponds to the nested sampling class of nested estimation problems.

Keen eyed readers may have noticed that this use of **`conditional`** in `p1-sim` is distinct to elsewhere in the paper as we have explicitly used SMC inference with a provided number of particles $M$ for **`conditional`**. This provides a roundabout means of controlling the computational budget for calls to **`conditional`**, as we showed is required for convergence in Section 3. In Figure 4 we see that changing $M$ changes the probability of each player betting under the model. Increasing $M$ corresponds to P2's calculation being carried out more carefully and so it is perhaps not surprising that it becomes less beneficial for P1 to bet with a weak hand as $M$ increases. Nonetheless, even for a large $M$ it remains beneficial for P1 to bet with less than average hands, reflecting the fact that money is already invested in the pot through the blinds and the opportunity to win through bluffing.

## C Exact Sampling

It may in fact be possible to provide consistent estimates in many nested query problems without requiring infinite computation for each nested call by using exact sampling methods such as rejection sampling. Such an approach is taken by Church [7], wherein no sample ever returns until it passes its local acceptance criterion. Church is able to do this because it only allows for hard conditioning on events with non-zero measure, allowing it to take a guess-and-check process that produces an exact sample in finite time by simply sampling from the generative model until the condition is satisfied. Although the performance clearly gets exponentially worse with increased nesting for such setups, this is a change in the constant factor of the computation, not its scaling with the number of samples taken: generating a single exact sample of the distribution has a finite expected time using rejection sampling which is thus a constant factor in the convergence rate.

Unfortunately, most problems require conditioning on measure zero events because they include continuous data – they require a soft conditioning akin to the inclusion of a likelihood term – and so cannot be tackled using Church. Constructing a generic exact sampler for soft conditioning in an automated way is likely to be insurmountably problematic in practice. Nonetheless, it does open up the intriguing prospect of a hypothetical system that provides a standard Monte Carlo convergence rate for multiply-intractable distributions. This assertion is a somewhat startling result: it suggests that Monte Carlo estimates made using nested exact sampling methods have a fundamentally different convergence rate for nested inference problems (though not nested estimation problems in general) than, say, nested self-normalized importance sampling (SNIS). Amongst other things, this might lead, perhaps through combination with the work of [16], to a contradiction of the, ostensibly reasonable, conjecture made by [17] that there are no generic tractable MCMC schemes for doubly-intractable distributions. We leave formal analysis of this tantalizing line of inquiry to future work.

## D Experimental Design Example

Figure 5 shows generic Anglican code for carrying out Bayesian experimental design, providing a consistent means of carrying out this class of nested estimation problems. Figure 6 shows the convergence of this code with `prior` and `lik` setup to reflect the delay discounting model described in [24, Appendix I]. Also shown is the convergence (or more specifically lack there of) in the case where $M = N_1$ is held fixed and the superior convergence achieved when exploiting the finite number of possible outputs to produce a reformulated, standard Monte Carlo, estimator as detailed in [24]. It therefore highlights both the importance of increasing the number of samples used by the inner query and exploiting our outlined special cases when possible.

```
(defm prior [] (normal 0 1))
(defm lik [theta d] (normal theta d))

(defquery inner-q [y d]
 (let [theta (sample (prior))]
  (observe (lik theta d) y)))

(defn inner-E [y d M]
 (->> (doquery :importance
        inner-q [y d])
      (take M)
      log-marginal))

(with-primitive-procedures [inner-E]
 (defquery outer-q [d M]
  (let [theta (sample (prior))
        y (sample (lik theta d))
        log-lik (observe*
                  (lik theta d) y)
        log-marg (inner-E y d M)]
   (- log-lik log-marg))))

(defn outer-E [d M N]
 (->> (doquery :importance
        outer-q [d M])
      (take N)
      collect-results
      empirical-mean))
```

Figure 5: Anglican code for Bayesian experimental design. By changing the definitions of `prior` and `lik`, this code can be used as a NMC estimator (consistent as $N, M \to \infty$) for any static Bayesian experimental design problem. Here **`observe*`** is a function for returning the log likelihood (it does not affect the trace probability), **`log-marginal`** produces a marginal likelihood estimated from a collection of weighted samples, and **`->>`** successively applies a series of functions calls, using the result of one as the last input the next. When `outer-E` is invoked, this runs importance sampling on `outer-q`, which, in addition to carrying out its own computation, calls `inner-E`. This in turn invokes another inference over `inner-q`, such that a MC estimate using `M` samples is constructed for each sample of `outer-q`. Thus `log-marg` is MC estimate itself. The final return is the (weighted) empirical mean for the outputs of `outer-q`.

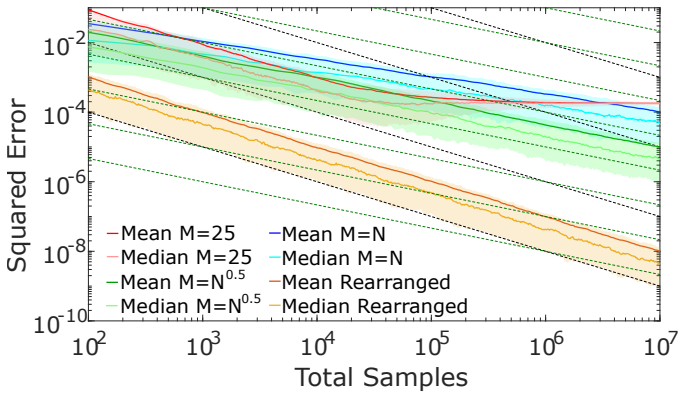



Figure 6: Convergence of NMC (i.e. code shown in Figure 5) and a reformulated standard Monte Carlo estimator for a Bayesian experimental design problem. Results are averaged over 1000 independent runs, while shaded regions give the 25%-75% quantiles, with a ground truth estimate made using a single run of the reformulated estimator with $10^{10}$ samples. We see that the theoretical convergence rates are observed (as indicated by the dashed lines), with the advantages of the reformulated estimator particularly pronounced.