Reversible Jump Probabilistic Programming

David A Roberts **Marcus Gallagher** **Thomas Taimre**
The University of Queensland, Australia

Abstract

In this paper we present a method for automatically deriving a Reversible Jump Markov chain Monte Carlo sampler from probabilistic programs that specify the target and proposal distributions. The main challenge in automatically deriving such an inference procedure, in comparison to deriving a generic Metropolis–Hastings sampler, is in calculating the Jacobian adjustment to the proposal acceptance ratio. To achieve this, our approach relies on the interaction of several different components, including automatic differentiation, transformation inversion, and optimised code generation. We also present Stochaskell, a new probabilistic programming language embedded in Haskell, which provides an implementation of our method.

1 INTRODUCTION

Probabilistic programming languages (PPLs) comprise two components: a language for concisely specifying probabilistic models, and a procedure for translating such a specification into an executable program capable of performing probabilistic inference. The task of inference is to compute the posterior distribution of model parameters given evidence, and often takes the form of drawing approximate samples from the posterior. By automating this translation, inference for models can be implemented more easily and with less risk of human error, compared to the manual derivation process required to implement inference procedures for probabilistic models in non-specialised programming languages. A wide variety of PPLs are currently available (Roy, 2018).

The Lightweight Metropolis–Hastings (M–H) framework (Wingate et al., 2011) is a popular option for implementing inference in PPLs. By combining the M–H algorithm with a simple proposal distribution, namely incrementally resampling from the prior model, Lightweight M–H can derive an inference procedure in a fully automated manner. However, such a simple proposal is often not the most efficient choice for performing inference on any given model.

One approach to alleviating the performance issues associated with M–H inference is to utilise additional information about the target distribution within proposals. A popular approach in this vein is the Hamiltonian Monte Carlo (HMC) method, which uses the gradient of the target distribution to guide proposals (Betancourt, 2017). However, the price of relying on gradients is that HMC is not applicable to models with discrete parameters. In particular, this precludes the ability to perform inference on trans-dimensional models, in which the dimensionality of the parameter space is itself an unknown parameter. In practice this means that users of PPLs relying on HMC must avoid such models entirely or, where possible, manually marginalise such parameters out of the model prior to implementation (Stan Development Team, 2017, §15).

A promising alternative approach to addressing the performance of M–H inference in PPLs is to allow users to specify custom proposal distributions in the form of a probabilistic program (Cusumano-Towner and Mansinghka, 2018). This avoids HMC's limitations on discrete parameters, whilst allowing the user to improve the efficiency of the inference procedure by supplying their own domain-specific knowledge. The trade-off is that inference is no longer *fully* automated, as users must supply a proposal program, but it is still possible to automate much of the work that would be required to implement this procedure without the aid of a PPL.

A deeper limitation is that, in its usual formulation, M–H is only applicable to models in which the dimensionality of the parameter space is constant. Although this limitation can be safely ignored in some circumstances, such as in the case of Lightweight M–H (Wingate et al., 2011, p. 773), in general doing so can

Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics (AISTATS) 2019, Naha, Okinawa, Japan. PMLR: Volume 89. Copyright 2019 by the author(s).

lead to the inference procedure producing incorrect results (Ranca and Ghahramani, 2015, §3.2.1). To address this limitation, we can turn to Reversible Jump Markov chain Monte Carlo (RJMCMC), a well-known reformulation of M–H that can safely perform inference on trans-dimensional models (Green and Hastie, 2009).

We will begin by briefly discussing the necessary background knowledge on RJMCMC. Following this we present our PPL, Stochaskell, focusing on the features it provides that aid in implementing our approach to automatically deriving RJMCMC samplers. We then provide a high-level overview of our approach and its implementation in Stochaskell, before outlining the various components it relies upon. Finally, we demonstrate our approach by implementing RJMCMC inference for two standard example models, and conclude by discussing potential directions for future work.

2 BACKGROUND

The purpose of a probabilistic program is to unambiguously specify the joint distribution $p(x, y)$ of the hidden parameters (or state) x and observations y of a probabilistic model. The task of a PPL is to then infer the posterior distribution $p(x|y)$ given values for the observations. The posterior density is often intractable to compute, so we rely instead on methods that require only the computation of $f(x) = p(x, y)$ which is proportional to $p(x|y)$ for fixed y.

The M–H algorithm is a Markov chain Monte Carlo (MCMC) method often used for approximately sampling from posterior distributions. Samples are produced in an iterative manner, with each iteration involving two steps. The first step takes the most recent sample x, and generates a candidate for the next sample x^* according to a problem-specific proposal density $q(x^*|x)$. The second step is to decide whether to accept this candidate as the next sample, or to reject it and take the next sample to simply be a duplicate of the last. This decision is made randomly, accepting with probability $\min(1, \alpha)$ where the acceptance ratio α is defined as

$$\alpha = \frac{f(x^*)q(x|x^*)}{f(x)q(x^*|x)}\,.$$

Note the efficiency of the M–H method depends on selecting a suitable proposal distribution that makes large enough jumps in parameter space whilst keeping the acceptance ratio close to unity, in order to explore the target distribution as quickly as possible.

Building on the same framework as M–H, RJMCMC begins by making some additional assumptions about the structure of the proposal distribution:

- For an initial state x, it should be possible to deconstruct the proposal distribution into a two step process: sampling an auxiliary variable u from some density $g(\cdot)$, and then calculating the proposed transition $x^* = h(u; x)$, where h is a deterministic function invertible in its first argument.
- Likewise, considering the reverse move, if the initial state had been x^*, the proposal would sample the auxiliary $u^* \sim g^*(\cdot)$ and calculate the deterministic transform $x = h(u^*; x^*)$.
- In order to be able to make the Jacobian adjustment to the acceptance ratio, the dimensionality of these auxiliary variables is constrained such that dimensionality of (x, u) equals that of (x^*, u^*). Either auxiliary variable may be absent, in which case we count its dimensionality as zero.

Fortunately proposals with this structure can be easily expressed as probabilistic programs. With these assumptions, RJMCMC defines the acceptance ratio

$$\alpha = \frac{f(x^*)g^*(u^*)}{f(x)g(u)}\left|\frac{\partial(x^*, u^*)}{\partial(x, u)}\right| \tag{1}$$

where the latter term is the absolute value of the determinant of the Jacobian matrix of (x^*, u^*) differentiated with respect to (x, u). This is easily extended to handle discrete parameters, by including their associated probabilities in the ratio but leaving them absent from the Jacobian (Green and Hastie, 2009).

Note that u^* is not directly observed, but instead needs to be calculated from x and x^* by inverting the transformation $x = h(u^*; x^*)$ to obtain $u^* = h^{-1}(x; x^*)$. This will be discussed in further detail in section 4.

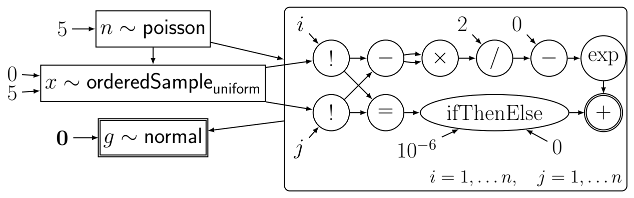
3 STOCHASKELL

Here we present Stochaskell, a new domain-specific language for probabilistic programming, embedded in Haskell. Stochaskell programs are simply regular Haskell programs, augmented with a library of data types and functions which allow probabilistic models to be conveniently constructed. This embedded approach, as opposed to creating a standalone language with its own syntax, allows us to immediately take advantage of the host language's type system, existing libraries, build tools, capacity for abstraction and modularity, and other general-purpose facilities (Kiselyov and Shan, 2009). Haskell is a particularly convenient host language to use in this context, as its first-class support for monads makes it easy to express probabilistic programs (Ścibior et al., 2015). Code and documentation for Stochaskell is freely available.[1]

[1]`https://davidar.github.io/stochaskell`

```
program :: P RVec
program = do
  n ← poisson 5
  let base = uniform 0 5
  x ← orderedSample n base :: P RVec
  let mu = vector [ 0 | i ← 1...n ]
      kernel a b = exp (−(a − b)^2 / 2)
           + if a == b then 1e−6 else 0
      cov = matrix
        [ kernel (x!i) (x!j)
        | i ← 1...n, j ← 1...n ]
  g ← normal mu cov
  return g
```

(a)



(b)

Figure 1: (a) A simple probabilistic program written in Stochaskell, with (b) an illustration of the corresponding intermediate representation. The rounded rectangle represents the cov matrix, with the double circle indicating the root node which represents the value of the matrix entry for any given values of i and j. The rectangles represent the values of random variables, with the double rectangle being the return value of the program.

3.1 Intermediate Representation

We will now provide a brief overview of Stochaskell's intermediate representation (IR). The IR follows the structure of the probabilistic programs written by users in Stochaskell fairly closely, but with a focus on enabling it to be consumed and produced by other programs. In particular, our method for automatically deriving RJMCMC samplers operates upon this IR. The IR consists of two main components — expressions and programs — which will now be considered in turn. Figure 1 illustrates the IR of a simple program chosen to illustrate several features of the IR.

Stochaskell expressions are written as regular Haskell expressions. However, rather than producing a concrete numeric value, these expressions evaluate to a symbolic representation, stored as a directed acyclic graph (DAG). These graphs consist of several kinds of nodes, each representing a value, with incoming edges connecting them to the nodes which represent their inputs (if any). The following kinds of nodes are available:

Var external variable, e.g. the value of a random variable or an array index

Const constant value, from scalars through to n-dimensional concrete arrays

Data combines a variant tag and zero or more fields into an instance of an algebraic data type (ADT)

BlockArray generalisation of block matrices to n-dimensional arrays

Index application of the subscript operator (!)

Extract retrieves a specific field from an ADT

Cond conditional reference, associates a set of mutually exclusive conditions with a value to take whenever the corresponding condition is true

Apply application of a mathematical operation or primitive function

Array generated by abstract array definitions, e.g. cov in Fig. 1

FoldScan applications of the fold and scan higher-order functions

Case case expression, which destructures an ADT and evaluates a different expression per tag

Each DAG represents a single scope, and nested scopes (such as those created by abstract array definitions) are represented by the sequence of corresponding DAGs, called a *Block*. Although a DAG can belong to a number of different Blocks, its position in the sequence is always the same — as the sequence of parent scopes cannot change, due to lexical scoping — which we call the *level* of the DAG. Edges can connect nodes from different DAGs in a Block, but can only be directed from parent scopes to child scopes, not vice versa. An *Expression* is defined as a function that takes an existing Block of definitions, and returns an updated Block containing any additional nodes required by the expression, along with a reference to the root node representing the value of the whole expression.

Probabilistic programs are represented as a sequence of stochastic choices, linked by a graph of deterministic transformations (represented by a Block). Each of these is either a sample from a primitive distribution (e.g. normal), or a random array with marginals specified by primitive distributions. Distributions are associated with a list of references to the nodes representing the relevant parameters. They may also take another primitive distribution as a parameter, as in the case of orderedSample. To represent conditional distributions, each random variable can optionally be associated with a concrete value corresponding to the conditioned value. Finally, this representation is wrapped within Haskell's *State* monad, to define the *P* monad visible in the type signatures of the programs presented in this paper.

3.2 Code Generation

As mentioned in section 3.1, the IR is designed to be a representation of probabilistic programs that can be consumed and transformed by other programs. One particularly useful kind of transformation is provided by code generators, which transform the IR into code that can be used with an external (probabilistic) programming language. This allows users to write a model once in Stochaskell, and apply multiple probabilistic programming systems to various parts of it. Stochaskell currently provides code generators targeting several different PPLs, including Stan (Carpenter et al., 2017), PyMC3 (Salvatier et al., 2016), Edward (Tran et al., 2017), and Church (Goodman et al., 2008). A C++ code generator is also provided, for compiling the IR to optimised machine code.

In section 5 we exploit this to perform inference on a model implemented in Stochaskell via a combination of our RJMCMC approach alongside HMC provided by Stan. This allows us to provide high-performance inference for (conditionally) fixed-dimensional, continuous model parameters, whilst maintaining the ability to support trans-dimensional and discrete model parameters via RJMCMC.

4 APPROACH

This section outlines our approach to automatically deriving RJMCMC inference procedures. We require the user to supply two probabilistic programs, specifying a target and proposal distribution. Using these, our method produces a third probabilistic program, implementing the transition kernel of the reversible jump Markov chain.

We have implemented our method within the Stochaskell PPL, where it is exposed to the user via the rjmc operator. In combination with the C++ code generation backend discussed earlier, this allows us to easily perform RJMCMC inference:

```
let transition x =
  target `rjmc` proposal `runCC` x
samples ← iterateLimit 1000 transition x0
```

At a high level, the implementation of this operator is quite simple, encoding the structure of a RJMCMC transition into a probabilistic program:

```
rjmc target proposal x = do
  let a = rjmcRatio target proposal
  y ← proposal x
  accept ← bernoulli (min' 1 (a x y))
  return (if accept then y else x)
```

Most of the complexity here is hidden inside the function rjmcRatio, which produces a deterministic program (as a Stochaskell expression) for computing the RJMCMC acceptance ratio (1). Implementing this function requires a number of components, which we describe in the following subsections.

4.1 Automatic Gradient Computation

There are a number of different techniques for computing the gradients of numeric functions implemented as computer programs. *Numerical differentiation* methods achieve this by approximating the derivative via a finite difference calculation, whereas *automatic differentiation* and *symbolic differentiation* transform the program into one which implements the exact derivative (Baydin et al., 2018).

We follow the approach of Theano (Theano Development Team, 2016) and TensorFlow (Abadi et al., 2016). We represent probabilistic programs as a computation graph, in the form of the Stochaskell IR described earlier. Computing the derivative of an expression with respect to one of its variables can be achieved by walking through this graph. We first consider the root node, and apply the chain rule to decompose the derivative into a function of the derivatives of the dependencies of this node. This process continues recursively until reaching nodes with no dependencies, whose derivative is constant. Composing all of these partial derivatives yields a new IR graph, representing the required derivative. Tuples of values can also be differentiated, in which case the result is a block matrix containing the individual Jacobian matrices.

4.2 Transformation Inversion

As discussed earlier, we also need to be able to invert the deterministic transformation $x = h(u^*; x^*)$ within the proposal probabilistic program, to obtain a deterministic program that computes the auxiliary variable $u^* = h^{-1}(x; x^*)$. For this we utilise a simple equation solver.

The solver operates by traversing the computation graph in a similar manner to the automatic differentiator. Equations are recursively rewritten according to a list of rules for each primitive transformation. Most of these rules are quite straightforward, such as rewriting $e^z = c$ to $z = \log c$. In this case, if the left hand side z is a random variable, we record this equation as its solution. Otherwise, we substitute the definition of z into the equation and apply another rewrite rule.

Note that this works with not just mathematical expressions, but also with more general data manipulation functions. For example, to solve $v\,\mathrm{insertAt}(i, e) = c$ (where the left hand side is the vector v with an element e inserted at index i), we find the first index j

Algorithm 1 Rule for solving equations of the form (case i of $1 \to e_1; 2 \to e_2; \dots n \to e_n) = c$

```
let $\theta_j$ contain the solutions of $e_j = c$, for $j = 1, \dots n$
let $\varphi_j = \theta_j$ restricted to solutions of program inputs
for $k = 1, \dots n$ do
    condition solutions in $\theta_k$ on "$i = k$"
    if $\varphi_k$ is not empty and $\varphi_k \neq \varphi_j$ for $j \neq k$ then
        solve $i = k$, conditioning solutions on:
            "all equations in $\varphi_k$ must be satisfied"
return all the conditioned solutions
```

where $v_j \neq c_j$, then solve both $i = j$ and $e = c_j$.

In the case of binary operators, inversion is performed only with respect to the operand with unknown value. So, to solve $a + b = c$ when a already has a solution, we rewrite the equation to $b = c - a$.

The most complex rule is for dealing with case expressions, outlined in Algorithm 1. The conditioned solutions returned by this are later combined to form conditional reference nodes.

The main limitation of this (relatively simple) method is that it is unable to solve systems of simultaneous equations. As such, proposals should be written so that each component of the proposed state introduces at most one auxiliary variable. That is, we require $h(\mathbf{u}; x)$ to be represented in the following form:

$$h(\mathbf{u}; x) = (h_1(u_1; x), h_2(u_1, u_2; x), \dots, h_n(\mathbf{u}; x)).$$

4.3 Code Optimisation

Programs that have been automatically generated by procedures such as those described above often contain many redundant and inefficient computations. Passing these programs unmodified through to the code generators would therefore result in poor performance when they are eventually executed. To avoid this, it is necessary to apply a number of optimisation passes to programs prior to code generation.

We have implemented several such code optimisations in Stochaskell to address major performance issues. The first set of optimisations are automatically applied to all Stochaskell programs as they are constructed. In particular, common subexpression elimination is performed using the *hash-consing* technique described by Kiselyov (2011). Constant floating is also performed to lift nodes into parent scopes whenever possible, which improves the efficiency of generated code by avoiding unnecessary recomputation.

Various basic arithmetic simplifications are also performed, such as immediately evaluating constant expressions, removing identity operations (e.g. multiplication by one or applying a function to its inverse), flattening nested conjunctions (as well as removing duplicate terms and detecting contradictions), and expanding determinants of products. At this stage conditional references are also propagated through the computation graph so that they can be combined and simplified (e.g. removing conditions known to be false).

In addition to this, there is a set of optional optimisations that need to be explicitly triggered. These are mostly related to conditional references, block matrices, and combinations thereof. In particular, nested conditional references are flattened (including cases where conditions are themselves conditional references), conditional references are lifted out of block matrices (discussed further below), matrix operations on block matrices are expanded in terms of operations on individual blocks, and determinants of block triangular matrices are optimised by evaluating only determinants on the diagonal.

4.4 Reversible Jump Jacobian

To compute the Jacobian term in the RJMCMC acceptance ratio (1), we begin by expressing it in a more direct manner. Substituting $x^* = h(u; x)$ and $u^* = h^{-1}(x; x^*)$, and applying the chain rule yields

$$\left| \begin{matrix} \frac{\partial h(u;x)}{\partial x} & \frac{\partial h(u;x)}{\partial u} \\ \frac{\partial h^{-1}(x;x^*)}{\partial x} + \frac{\partial h^{-1}(x;x^*)}{\partial x^*} \frac{\partial h(u;x)}{\partial x} & \frac{\partial h^{-1}(x;x^*)}{\partial x^*} \frac{\partial h(u;x)}{\partial u} \end{matrix} \right|. \quad (2)$$

A symbolic expression for $h^{-1}(x; x^*)$ can be obtained by solving $x = h(u^*; x^*)$ for u^* using the equation solver outlined in section 4.2. Each of the derivatives within (2) can then be obtained by applying the automatic differentiator, and combined into a block matrix.

It is common for proposal distributions to be expressed as a mixture of sub-proposals, in which one of the sub-proposals is randomly selected on each step according to some probability distribution. A proposal might include multiple auxiliary variables with only a subset of them being used by each sub-proposal, in which case the Jacobian for each sub-proposal would include rows and columns for only those auxiliary variables it uses.

It is worth describing how the components of our procedure interact when encountering proposals with this structure. When the solver is invoked on $u^* = h^{-1}(x; x^*)$, the case expression used in the proposal to delegate to sub-proposals results in conditional references being returned for the auxiliary variables. These conditional references indicate which of the sub-proposals each auxiliary is used in, along with the value each auxiliary variable takes in each case. These conditional references are propagated through to the final Jacobian, resulting in a block matrix filled with conditional references. The code optimiser is then triggered to transform the Jacobian to a single conditional reference which delegates to Jacobians for each sub-proposal. This is achieved by selecting the appropriate value of the blocks for each condition, or removing blocks when the associated auxiliary is unconstrained by that condition.

4.5 Probability Density Functions

In addition to the Jacobian, the RJMCMC acceptance ratio (1) involves calculating the probability density function associated with the target distribution. In the simplest case, a probabilistic program may sample a sequence of auxiliary random variables from primitive distributions, returning them directly without applying any transformation. In this case, the joint density can be easily obtained by taking the product of the densities of the individual primitive distributions.

In general, probabilistic programs may instead return a (deterministic) transformation of the auxiliary variables, $z = t(u)$. As such transformations can be arbitrarily complicated, it would be infeasible to expect to be able to automatically compute the resulting probability density for all possible probabilistic programs (which may involve intractable integrals). Therefore, we will restrict our focus to the case where the transformation is invertible. In this case, the probability density of the whole program can be expressed in terms of the joint density of the auxiliary variables $g(u)$ along with a Jacobian adjustment: $p(z) = g(u)/\left|\frac{\partial z}{\partial u}\right|$ (Kroese et al., 2011, §A.6). As in the previous section, we can utilise the transform inverter to determine the value of the auxiliary variables $u = t^{-1}(z)$, and the automatic differentiator to calculate the Jacobian.

5 EXPERIMENTS

Here we demonstrate how our approach can be used to easily implement RJMCMC inference for two standard example models. For each, both the generative model and proposal distribution are written as Stochaskell programs,[2] which are then used to automatically derive a RJMCMC sampler over the posterior.

5.1 Soccer Goals

Figure 2 provides an implementation of the simple model described in Green and Hastie (2009, §3.1). This trans-dimensional model is a model choice problem with two candidate models. Model 1 distributes

[2]Note that the proposal is specified as a single probabilistic program, which Stochaskell automatically decomposes into the required auxiliary distribution $g(u)$ and deterministic transformation $h(u; x)$.

```
soccer1 :: Z → P (R,ZVec)
soccer1 n = do
  lam ← gamma 25 10
  y ← joint vector [ poisson lam
                   | _ ← 1...n ]
  return (lam,y)

soccer2 :: Z → P (R,R,ZVec)
soccer2 n = do
  lam ← gamma 25 10
  kap ← gamma 1 10
  let a = 1/kap
      b = a/lam
  y ← joint vector [ negBinomial a b
                   | _ ← 1...n ]
  return (lam,kap,y)

data Model = Model1 R ZVec
           | Model2 R R ZVec
soccer :: Z → P (Expr Model)
soccer n = do
  (lam1,      y1) ← soccer1 n
  (lam2,kap2,y2) ← soccer2 n
  k ← bernoulli 0.5
  return $ if k
    then fromConcrete (Model1 lam1 y1)
    else fromConcrete (Model2 lam2 kap2 y2)

soccerJump :: Model → P (Expr Model)
soccerJump (Model1 lam y) = do
  u ← normal 0 1.5
  let kap = 0.015 ∗ exp u
  return $ fromConcrete (Model2 lam kap y)
soccerJump (Model2 lam _ y) =
  return $ fromConcrete (Model1 lam y)

soccerHMC :: Z → Model → IO Model
soccerHMC n (Model1 lam y) = do
  let posterior =
        [ lam'
        | (lam',y') ← soccer1 n
        , y' == y ]
  samples ← hmcStanInit 10 posterior lam
  let lam' = last samples
  return (Model1 lam' y)
soccerHMC n (Model2 lam kap y) = do
  let posterior =
        [ (lam',kap')
        | (lam',kap',y') ← soccer2 n
        , y' == y ]
  let s0 = (lam,kap)
  samples ← hmcStanInit 10 posterior s0
  let (lam',kap') = last samples
  return (Model2 lam' kap' y)

soccerStep :: Z → Model → IO Model
soccerStep n m = do
  m' ← soccerHMC n m
  m'' ← soccer n `rjmcC` soccerJump
          `runCC` fromConcrete m'
  return $ fromRight (eval' m'')
```

Figure 2: Stochaskell implementation of the soccer goal model from Green and Hastie (2009).

the observed data y according to a Poisson distribution, as defined by the probabilistic program soccer1. Model 2 instead distributes the data according to a negative binomial distribution, as shown in soccer2. These candidate models are then combined into the full trans-dimensional model soccer.

The proposal distribution has two parts, depending on whether the current state of the Markov chain is in model 1 or 2. In the former case, Green and Hastie (2009) propose a jump from model 1 to 2 by sampling an auxiliary variable $u \sim \mathcal{N}(0, 1.5)$ and transforming the model parameters as $\lambda \mapsto (\lambda, 0.015e^u)$. The reverse proposal simply discards a parameter, as $(\lambda, \kappa) \mapsto \lambda$. The combined proposal is transcribed to the probabilistic program soccerJump.

This takes care of proposing jumps between models 1 and 2, but we still need to define within-model jumps to complete the inference procedure. Green and Hastie (2009) do not specify any particular method for doing this, so we simply offload this task to the Stan code generation backend, as shown in soccerHMC.[3] Finally, we combine all these elements to produce the Markov chain transition kernel soccerStep.

Note that nowhere have we had to concern ourselves with the details of implementing the RJMCMC inference procedure; this has all been taken care of automatically by Stochaskell as described in section 4.

To test this model, we use the total soccer goal data from Curley (2016). Performing inference by running iterateLimit 5000 (soccerStep n) (Model1 1 goalData) yields a posterior probability of 70.76% in favour of model 1, in agreement with Green and Hastie (2009).

5.2 Coal Mining Disasters

We now consider a more complex model, namely the coal mining disaster model introduced in the seminal RJMCMC paper (Green, 1995). The model partitions a time series into a number of segments separated by n change-points at locations $(s_i)_{i=1,\dots n}$, with $s_0 = 0$ and $s_{n+1} = t$ as fixed endpoints. The data is modelled by a Poisson process, with rate g_j between each pair of change-points (s_{j-1}, s_j). This is encoded in the probabilistic program coalLikelihood in Fig. 3.

The number of change-points n is assumed to have a truncated Poisson prior distribution, their locations $(s_i)_{i=1,\dots n}$ to be the even-numbered order statistics of the uniform distribution over the interval $[0, t]$, and the

```
coalLikelihood :: R→(Z,RVec,RVec)→P RVec
coalLikelihood t (n,s,g) = do
  let rate y = g!(findSortedInsertIndex y s)
      widths = vector
        [ (s!(i+1)) - (s!i)
        | i ← 1...(n-1) ] :: RVec
      w = blockVector [cast (s!1), widths,
        cast (t - (s!n))] :: RVec
      areas = vector [ (w!j) * (g!j)
                     | j ← 1...(n+1) ]
      area = sum' areas
  y ← poissonProcess t rate area
  return y

coalPrior :: R → P (Z,RVec,RVec)
coalPrior t = do
  let lam = 3; nMax = 30; a = 1; b = 200
  n ← truncated 1 nMax (poisson lam)
  s' ← orderedSample
         (2*n + 1) (uniform 0 t) :: P RVec
  let s = vector [ s'!(2*i) | i ← 1...n ]
  g ← joint vector [ gamma a b
                   | _ ← 1...(n+1) ]
  return (n,s,g)

type Model = (Z,RVec,RVec,RVec)
coal :: R → P Model
coal t = do
  (n,s,g) ← coalPrior t
  y ← coalLikelihood t (n,s,g)
  return (n,s,g,y)
```

Figure 3: Stochaskell implementation of the coal mining disaster model from Green (1995).

rates $(g_j)_{j=1,\dots n+1}$ to be iid Gamma distributed. This is expressed by the probabilistic program coalPrior. Combining these yields the full generative model coal.

There are four sub-proposals implemented in Fig. 4: two for updating individual change-point locations and rates, and two for incrementing or decrementing the number of change-points. The former two are quite straightforward, as expressed by the programs coalMovePoint and coalMoveRate. The latter two sub-proposals, as used by Green (1995, pp. 720–721), are a little more involved. In the interests of simplicity, we will use slightly different sub-proposals than those described in the original paper, as specified by coalMoveBirth and coalMoveDeath.

We then define the function coalMoveProbs (not shown due to space considerations) to calculate the probability of selecting each sub-proposal, derived from the constraints described in Green (1995, p. 719). Finally we encode the full proposal distribution in coalMove, and the RJMCMC transition kernel is automatically derived in coalStep. Figure 5 shows the result of running this model on the *coal* dataset provided by the *boot* package for R.

[3] It should also be possible to perform the HMC update *within* the RJMCMC proposal using our approach combined with the mechanism proposed by Al-Awadhi et al. (2004), which involves only minimal modifications to the acceptance ratio (1).

```
coalMovePoint :: R → Model → P Model
coalMovePoint t (n,s_,g,y) = do
  let s i = if i == 0 then 0
        else if i == (n+1) then t
        else (s_!i)
  j ← uniform 1 n
  u ← uniform 0 1
  let sj' = s (j−1) +
           (s (j+1) − s (j−1)) ∗ u
      s' = s_ `replaceAt` (j, sj')
  return (n,s',g,y)

coalMoveRate :: R → Model → P Model
coalMoveRate t (n,s,g,y) = do
  j ← uniform 1 (n+1)
  u ← uniform (−1/2) (1/2)
  let gj = exp (log (g!j) + u)
      g' = g `replaceAt` (j, gj)
  return (n,s,g',y)

coalMoveBirth :: R → Model → P Model
coalMoveBirth t (n,s,g,y) = do
  x ← uniform 0 t
  let j = findSortedInsertIndex x s
      s' = s `insertAt` (j, x)
  u ← lognormal 0 1
  let g' = g `insertAt` (j+1, u ∗ (g!j))
  return (n+1,s',g',y)

coalMoveDeath :: R → Model → P Model
coalMoveDeath t (n,s,g,y) = do
  j ← uniform 1 n
  let s' = s `deleteAt` j
      g' = g `deleteAt` (j+1)
  return (n−1,s',g',y)

coalMove :: R → Model → P Model
coalMove t m@(n,_,_,_) = do
  let (e,p,b,d) = coalMoveProbs n
  mixture' [(e, coalMoveRate  t m)
           ,(p, coalMovePoint t m)
           ,(b, coalMoveBirth t m)
           ,(d, coalMoveDeath t m) ]

coalStep :: R → Model → IO Model
coalStep t m =
  coal t `rjmc` coalMove t `runStep` m
```

Figure 4: Inference for the model in Fig. 3.

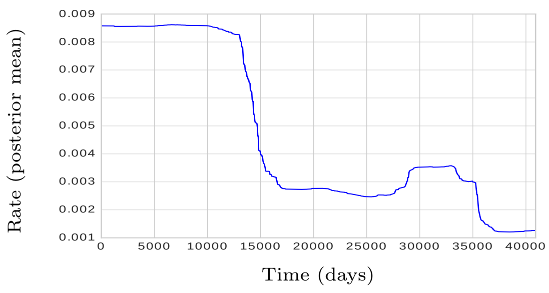


Figure 5: Result of running the inference procedure in Fig. 4, in agreement with Green (1995, Fig. 1).

6 DISCUSSION

We have presented a method for automatically deriving RJMCMC inference procedures when provided with the target and proposal probabilistic programs. We have also presented Stochaskell, a new PPL embedded in Haskell, which provides an implementation of this method. Stochaskell's IR also enables other applications which rely on transforming probabilistic programs, such as combining inference methods provided by multiple PPLs to be used on a single model. We demonstrated how this can be used to concisely implement RJMCMC for models, whilst being powerful enough to handle a realistic problem.

Integrating RJMCMC inference with PPLs has been previously investigated. For instance, it is possible to implement RJMCMC samplers within the NIMBLE PPL, however this requires the user to manually calculate RJMCMC acceptance ratios and implement the associated bookkeeping themselves (NIMBLE Development Team, 2017). Narayanan and Shan (2018) have recently presented an extension to the disintegration method utilised by the Hakaru PPL that allows it to compute acceptance ratios from RJMCMC proposal distributions implemented as probabilistic programs. However, they note that it is still necessary for users to manually specify an additional *base distribution* for each proposal distribution. To the best of our knowledge, the implementation of our method in Stochaskell is the first to provide automatic derivation of a RJMCMC inference procedure, when provided with only probabilistic programs implementing the generative model and proposal distribution.

We have demonstrated that our method can be applied to realistic models with minimal manual effort, but there is further work that needs to be done to ensure that the computational efficiency of RJMCMC samplers automatically produced by our method is competitive with that of handwritten samplers. As discussed in section 4.3, the main concern is in optimising away inefficiencies present in automatically generated code. We have outlined the optimisation passes we have already implemented to address some of these, but further work will be required to close this gap.

Another direction for future work is to investigate alternative methods for automatically inverting transformations of random variables. Although the simple method we describe in section 4.2 works well for a variety of transformations, including those necessary to implement realistic proposal distributions, it does not handle arbitrarily complex transformations. As such, it may be worthwhile investigating the integration of more sophisticated formalisations, such as the *preimage* semantics introduced by Toronto et al. (2015).

Acknowledgements

We thank the anonymous reviewers for providing helpful recommendations regarding the presentation of the paper. This research was supported by an Australian Government Research Training Program Scholarship.

References

M. Abadi et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

F. Al-Awadhi, M. Hurn, and C. Jennison. Improving the acceptance rate of reversible jump MCMC proposals. *Statistics & Probability Letters*, 69(2):189–198, Aug. 2004. ISSN 0167-7152. doi: 10.1016/j.spl.2004.06.025.

A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic Differentiation in Machine Learning: A Survey. *Journal of Machine Learning Research*, 18(153):1–43, 2018. URL `http://www.jmlr.org/papers/v18/17-468.html`.

M. Betancourt. The Convergence of Markov chain Monte Carlo Methods: From the Metropolis method to Hamiltonian Monte Carlo. *arXiv:1706.01520*, June 2017. URL `http://arxiv.org/abs/1706.01520`.

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017. ISSN 1548-7660. doi: 10.18637/jss.v076.i01.

J. P. Curley. Engsoccerdata: English Soccer Data 1871-2015, 2016. URL `https://cran.r-project.org/web/packages/engsoccerdata/README.html`.

M. F. Cusumano-Towner and V. K. Mansinghka. Using probabilistic programs as proposals. *arXiv:1801.03612*, Jan. 2018. URL `http://arxiv.org/abs/1801.03612`.

N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229, 2008. URL `https://arxiv.org/abs/1206.3255`.

P. J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, Dec. 1995. ISSN 0006-3444. doi: 10.1093/biomet/82.4.711.

P. J. Green and D. I. Hastie. Reversible jump MCMC. Technical report, June 2009. URL `https://people.maths.bris.ac.uk/~mapjg/papers/rjmcmc_20090613.pdf`.

O. Kiselyov. Implementing explicit and finding implicit sharing in embedded DSLs. *Electronic Proceedings in Theoretical Computer Science*, 66:210–225, 2011. ISSN 2075-2180. doi: 10.4204/EPTCS.66.11.

O. Kiselyov and C.-c. Shan. Embedded probabilistic programming. In *Domain-Specific Languages*, pages 360–384. Springer, 2009. doi: 10.1007/978-3-642-03034-5_17.

D. P. Kroese, T. Taimre, and Z. I. Botev. *Handbook of Monte Carlo Methods*. Wiley Series in Probability and Statistics. John Wiley and Sons, New York, New York, 2011. ISBN 978-0-470-17793-8.

P. Narayanan and C.-c. Shan. More support for symbolic disintegration, 2018. URL `https://pps2018.sice.indiana.edu/files/2017/12/abstract.pdf`.

NIMBLE Development Team. Writing reversible jump MCMC in NIMBLE, Feb. 2017. URL `https://r-nimble.org/writing-reversible-jump-mcmc-in-nimble`.

R. Ranca and Z. Ghahramani. Slice Sampling for Probabilistic Programming. *arXiv:1501.04684*, Jan. 2015. URL `http://arxiv.org/abs/1501.04684`.

D. Roy. Probabilistic programming, 2018. URL `http://probabilistic-programming.org/`.

J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, Apr. 2016. ISSN 2376-5992. doi: 10.7717/peerj-cs.55.

A. Ścibior, Z. Ghahramani, and A. D. Gordon. Practical probabilistic programming with monads. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, pages 165–176. ACM Press, 2015. ISBN 978-1-4503-3808-0. doi: 10.1145/2804302.2804317.

Stan Development Team. Stan modeling language users guide and reference manual, 2017. URL `http://mc-stan.org`. Version 2.17.0.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv:1605.02688*, May 2016. URL `http://arxiv.org/abs/1605.02688`.

N. Toronto, J. McCarthy, and D. Van Horn. Running Probabilistic Programs Backwards. In *Programming Languages and Systems*, pages 53–79. Springer, 2015. doi: 10.1007/978-3-662-46669-8_3.

D. Tran, M. D. Hoffman, R. A. Saurous, E. Brevdo, K. Murphy, and D. M. Blei. Deep Probabilistic Programming. *arXiv:1701.03757*, Jan. 2017. URL `http://arxiv.org/abs/1701.03757`.

D. Wingate, A. Stuhlmueller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011. URL `http://jmlr.org/proceedings/papers/v15/wingate11a.html`.