PROBABILISTIC PROGRAMMING LANGUAGES?

Maxwell Clarke – maxeonyx@gmail.com – 300559026

COMP440 Essay – Tue 22 Jun 2021 – Victoria University of Wellington

ABSTRACT

Probabilistic programming languages (PPLs) promise easy and rapid development of complex statistical models, with less expert knowledge, by writing them as stochastic programs and performing Bayesian inference on them.

Two major trade-offs have strongly influenced PPL design.

The first is whether or not to support inference algorithms that require gradient computation. Inference algorithms such Hamiltonian Monte Carlo (2.1.5) and NUTS (2.1.6) perform better than other methods such as Metropolis Hastings (2.1.3), but they require the gradient, which restricts the variable types that can be used in the programming language. Historically, languages such as Stan [1] did not support discrete variables in models. Now, Pyro [2] and Turing.jl [3] allow this to a limited extent using modern automatic differentiation. Recent languages such as Venture [4] are attempting to address this trade-off further.

The second is whether the programming language is standalone or embedded in a host language as a library. Standalone languages have had higher performance than libraries by being natively compiled, but they are harder to inter-operate with existing code. Recent languages such as NumPyro [5] for Python and Turing.jl [3] for Julia [6] have lessened this trade-off, by improving the performance of PPL libraries.

Future work in PPLs should focus on fixing bugs in existing languages, improving the inference algorithms, and improving user discovery of the trade-offs of different inference algorithms.

Contents

1. INTRODUCTION

I came into this course with little knowledge of probabilistic programming languages (PPLs), but with a number of questions, which I answer in this essay.

1. What are probabilistic programming languages?
2. What are the limitations of PPLs?
3. How do PPLs work?
4. What are the differences between different languages?
5. How promising is PPL research?
6. Should you be using PPLs?

In this section, I introduce the reader to probabilistic programming through a simple example, discuss what kinds of problem PPLs can solve, and what kinds of problem they cannot solve.

In Section 2 (Background) I discuss the different Bayesian inference methods that underlie PPLs, and some trade-offs that these methods give rise to.

In Section 3 (Comparison), I introduce the variety of PPLs that exist, and identify some key differences between them.

In Section 4 (Trade-offs), I discuss two main trade-offs that have shaped the development of PPLs and how they are being addressed, and how these trade-offs motivate future work in PPLs.

Lastly, I summarise by answering the question: Should you be using PPLs?

So, what are PPLs?

1.1. Probabilistic programming languages

Probabilistic programming languages (PPLs) are domain specific languages for defining probabilistic models. They allow concise definition of a model that can be used to sample from some distribution, called the prior. Importantly they then allow conditioning this distribution on some data or observations, to make a posterior distribution, which can also be sampled from.

The motivation behind probabilistic programming languages is:

- To make defining and performing inference on probabilistic models more ergonomic – improving development time, readability and correctness.
- To enable the creation of more complex models, by allowing user defined functions, and correctly supporting inference on programs with mixed and complex data types, unbounded loops, recursive functions, and different variables in different branches.

```
1  (defn observe-data [x y_obs slope bias]
2      (let [  y_pred (+ (* slope x) bias)
3              sigma 1.0
4              likelihood (normal y_pred sigma)]
5          (observe likelihood y_obs)))
6
7  (let [  slope (sample (normal 0.0 10.0))
8          bias  (sample (normal 0.0 10.0))
9          xdata [ 1.0
10                 2.0
11                 3.0
12                 4.0
13                 5.0
14                 6.0 ]
15         ydata [ 2.1
16                 3.9
17                 5.3
18                 7.7
19                 10.2
20                 12.9 ]]
21
22     (observe-data (get xdata 0) (get ydata 0) slope bias)
23     (observe-data (get xdata 1) (get ydata 1) slope bias)
24     (observe-data (get xdata 2) (get ydata 2) slope bias)
25     (observe-data (get xdata 3) (get ydata 3) slope bias)
26     (observe-data (get xdata 4) (get ydata 4) slope bias)
27     (observe-data (get xdata 5) (get ydata 5) slope bias)
28
29     [slope bias])
```

Listing 1. Simple linear regression model in ThisPPL

- To reduce the knowledge required to define and use probabilistic models and Bayesian inference algorithms.

One thing to note is that in this essay I refer to libraries for probabilistic programming as PPLs, since they perform very similar functions. I discuss the difference between the two in more detail in Section 4.2.

1.2. Example of a Probabilistic Program

In Listing 1 we see an example probabilistic program. It is a very simple model called Bayesian linear regression, in the programming language "ThisPPL" that I implemented for my assignments. This was based on the PPL described in "*An Introduction to Probabilistic Programming Languages*" [7].

The goal of the linear regression program is to fit a line to some data. This means learning the parameters of a linear relationship. However, instead of finding only a point estimate for the parameters, we will find and plot a probability distribution over the parameters. This will then allow us to visually understand how uncertain we are about the parameters.

The program starts at the *let* expression on line 7.

1. The two parameters *slope* and *bias* of the linear model are declared, both with Gaussian priors $\mathcal{N}(\mu = 0, \sigma = 10)$. Together, these variables form the prior of the program.
2. Two hard-coded lists of data are declared. These are the observations (data) which will be used to condition the model.

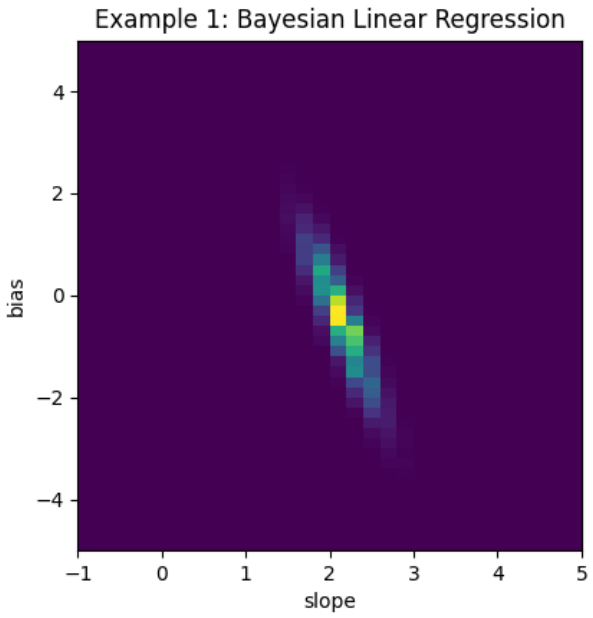

Fig. 1. Histogram of the posterior over *slope* and *bias* from Listing 1.

3. The user-defined function *observe data* is called once for each data point. This conditions the distribution of the parameters *slope* and *bias* on observing that data point.

4. The values of *slope* and *bias* are returned. This return expression simply defines the *query variables* of the program.

The function *observe data*, is where the program both defines the model and performs conditioning. First, the *slope* and *bias* are used to compute a predicted y value based on the observed x. The likelihood distribution $p(y|x)$ is then created with that predicted y as the mean, and observed to be the observed y value. In this model the likelihood can be thought of as modeling some independent Gaussian noise on y with $\sigma = 1$.

ThisPPL uses evaluation-based inference methods. This means that the program is run repeatedly, each time producing a sample. Running the program many times produces many samples, which form an empirical distribution (histogram) which approximates the posterior (also known as the target distribution) in the limit.

The resulting histogram from running this model is shown in Figure 1.

How this works in detail is discussed in more detail in Section 2. Now we will discuss what it is useful for.

1.3. Use cases of PPLs

PPLs perform Bayesian inference, which is used to find estimates of unknown parameters or variables when only sparse data is available, or when precise uncertainty estimates are necessary.

An example of a use case where Bayesian inference is used for precise uncertainty is in physics, such as for Mass Spectrometry in "Bayesian inference in physics: case studies" [8]. In this problem, we would like to know the chemical makeup of a sample given the measured mass spectrum. Using a PPL we can create a model which does the easier reverse problem – given a sample, simulates a mass spectrometer and produces a mass spectrum. We can then use Monte Carlo inference to infer a distribution over the concentration of chemical components that could result in the observed mass spectrum.

An example of a use case where Bayesian inference is used for when only sparse data is available is ecology [9]. The key useful part here is that Bayesian inference allows correctly incorporating existing known information into a model via the prior. An example from the above article is using Bayesian inference to find the best model explaining diversity of ant species in certain habitat types. They use MCMC in WinBUGS to perform model fitting in combination with Bayesian model comparison, on a dataset of ant species diversity.

The upside of Bayesian inference is that it will give the correct answer, given a valid prior and correct model. However, there are multiple downsides.

1.4. Limitations of Bayesian Inference

By far the biggest limitation of Bayesian inference is that it is computationally intensive, especially when estimating distributions over many variables. In order to estimate a distribution over some variable x, we need to perform and integral over the space of x, which I discuss in more detail in the next section. If x has a combinatorially large domain, or a continuous domain, then this integral can be intractable. Using approximate inference methods, we can perform approximate integrals in many of these situations. However, if the domain of the distribution is large enough then even these will not be enough, and Bayesian inference is simply not possible.

Another limitation is the difficulty of creating a prior. To perform Bayesian inference correctly on most problems, you need to choose an informative prior. This can be non-trivial even for simple cases. Take for example the simple example of a potentially biased coin. What is our prior on its flip probability θ? (For example, if you think someone is trying to trick you with a fake coin). I would say a multi-modal prior where we give weight to the beliefs that the coin is either biased strongly towards heads, or strongly towards tails, or not biased. How much density should there be in the other regions? How tight should the peaks be around 0, .5 and 1? To

get a useful answer, we need to answer these questions and construct such a distribution that represents our prior.

In the next section I introduce methods for approximating the integral in Bayesian inference.

2. BACKGROUND

In this section I go into detail on some PPL implementation techniques, such as various inference methods, and variable identifiers. Understanding these is useful for the later sections where I compare languages and discuss the trade-offs in PPL design.

2.1. Bayesian inference methods

Understanding the Bayesian inference methods that PPLs rely on is important, because some of the Bayesian inference methods have restrictions that affect the design of probabilistic programming languages which use them.

Examples of these restriction are whether an inference method requires transforming the program into a graph, or whether it requires computing the gradient of the program.

A good first method to understand is Importance Sampling.

2.1.1. Importance Sampling

Importance Sampling is a Monte Carlo method which samples from a distribution $\pi(x)$ by first sampling from a proposal distribution $q(x)$ and then weighting each sample x_i by its probability in the target = $\pi(x = x_i)$. This converges in the limit so long as $q(x)$ has compatible support with $\pi(x)$. Importance sampling is not just used for Bayesian inference - it can be used as a method to sample from any distribution where we can evaluate the density function.

In probabilistic programming, our target distribution is $p(x \mid y)$, and we can generate samples from the prior $p(x)$ by running the program. Sampling from the prior means not taking into account any observations y. We use these samples as our proposals, which means using the prior as our proposal distribution, so $q(x) = p(x)$. We then weight the samples x_i according to their probability in the posterior distribution $p(x = x_i \mid y)$.

Since we generate samples by running the program and do not need to derive any other representation of the program, this is called an *evaluation-based* inference method.

Importance sampling is a very flexible technique. It can work easily on any kind of program as long as we can evaluate the probability density function of the posterior. It is most appropriate for problems with a relatively uniform prior. On other problems, it may perform very badly. In particular, especially on high dimensional inference problems, the fact that importance sampling only "explores" by taking samples from the prior, means that it can fail to find regions of high probability in the posterior.

Markov chain methods perform better on this sort of problem. One such method is Gibbs Sampling.

2.1.2. Gibbs Sampling

Gibbs sampling is a graph-based Markov chain Monte Carlo method which repeatedly modifies a candidate or proposal solution (values for all the variables), and stores the values of the query variables at each step. In the limit, these samples converge to the posterior.

Gibbs sampling requires deriving a graph-based program representation, or graphical model. This may not be possible if the program has unbounded loops or recursion, which limits the use of Gibbs sampling to programs representable by a first-order probabilistic programming language.

Gibbs sampling requires the ability to sample from the distribution of a variable conditioned on the candidate values of all variables in its Markov Boundary on the graph. This is doable for discrete variables, but for continuous distributions or when it is intractable, Metropolis-Hastings can be used for this sampling step. This is then called Metropolis-in-Gibbs.

2.1.3. Metropolis-Hastings

Metropolis-Hastings (MH) [10], [11] is another Markov chain Monte Carlo method.

A Markov chain Monte Carlo (MCMC) method generates proposals by modifying the previous sample. Proposals are then either not accepted or they are accepted and become the new sample. Because of this process, the samples produced by an MCMC method are therefore not independent. Typically, some number of samples are skipped in order to get more independence.

Given a target distribution $\pi(x)$, Metropolis-Hastings works by sampling a proposal x_i' from a proposal distribution $q(x|x_{i-1})$, given the previous sample (accepted proposal) x_{i-1}. This proposal x_i' is then accepted as a sample or not based on an acceptance probability

$$\alpha(x_i', x_{i-1}) = \min\left(1, \frac{\pi(x_i')q(x_{i-1} \mid x_i')}{\pi(x_{i-1})q(x_i' \mid x_{i-1})}\right).$$

If the proposal x_i' is not accepted, a new $x_i' \sim q(x|x_{i-1})$ is drawn.

Similarly to Importance Sampling earlier, the proposal distribution can be an arbitrary distribution, subject to ergodicity. Informally, the repeated proposal process must be able to reach anywhere in the target distribution, and must never get stuck.

In probabilistic programming, drawing a new proposal means drawing new values for one or more of the random variables in the program. The target distribution $\pi(x)$ is as

before $p(x \mid y)$, and we must be able to evaluate the density of this function given x and y.

Metropolis-Hastings is easily used for purely continuous problems, using some distribution such as a multivariate Gaussian as the proposal distribution, and can easily be extended to problems with variables of any kind by using the prior as the proposal distribution. However, for programs with branching control flow, MH must be adapted.

One such adaptation is Single-Site Metropolis-Hastings (SSMH).

2.1.4. *Single-Site Metropolis Hastings*

When a probabilistic program has branching control flow, there may be different sets of variables in different branches of the control flow. In Importance Sampling this did not matter, because samples were drawn completely independently, and each run of the program simply takes one path through the branches. However, in MH we must remember the values of the variables in the previous sample in order to produce proposals based on our previous samples. How does this work when the previous sample had a different set of variables present?

Single-Site Metropolis Hastings [7] uses a two-step proposal to do correct MH on such programs. The proposal step is as follows:

- Given the set of variables domain(x_i) from the previously accepted sample x_i, pick one uniformly. This is the *single site* that will cause the samples to differ. We re-sample this variable given the state of the previous sample.
- For all other variables, if they are not dependent on the single site, keep their values from the previous sample.
- If they are dependent on the previous sample, re-sample them. At this point, because of the re-sampling, the program may take a different branch in the control flow and introduce variables that were not in the previous sample. These should be re-sampled.

At this point we have new values for all the variables and can evaluate the acceptance ratio as before.

The definition of *dependence* used here is important. When doing evaluation-based inference, an easy-to-implement definition of dependence is simply the order in the program. However, this is pessimistic and we may re-sample more than necessary, which will reduce the efficiency of the inference. Instead, we could run a dependency analysis on the program and use the true dependency ordering as the basis for re-sampling. This is done for example in [12], where they also extend the SSMH to retain the values (prevent resampling) of variables whose semantics (e.g. distribution type) did not change despite re-sampled dependencies.

Using an incorrect definition of dependence can lead to incorrect behaviour, which I discuss in Section 2.2.

SSMH shares the same downside as MH, which is that successive samples are correlated which decreases the efficiency of the sampling. Hamiltonian Monte Carlo addresses this problem specifically.

2.1.5. *Hamiltonian Monte Carlo*

Hamiltonian Monte Carlo (HMC) is a more sophisticated Markov Chain method than MH, with better efficiency due to a Hamiltonian dynamics heuristic which reduces correlation between samples.

However, HMC requires that we can compute or estimate the gradient of the Hamiltonian dynamical system with respect to the program variables. This has some consequences which I will discuss in Section 4.1.

2.1.6. *No U-Turn Sampler*

No U-Turn Sampler (NUTS) [13] is a variant on HMC which automatically tunes two parameters which were previously user-specified, which empirically performs at least as well as HMC and often better [13].

2.1.7. *Variational Inference*

Lastly, Variational Inference (VI) [14], [15] is a very different technique from the ones discussed above. In the above methods, we sample from an distribution that approximates the target in the limit. In SVI, we choose a distribution we can represent analytically, although it may not be the same form as the posterior. Unless we know the posterior is of the same form as our analytic distribution, VI is an asymptotically inexact approximation. We then optimize the parameters of this analytic distribution to be as close to the target as possible, according to some distance measure defined for analytic probability distributions - typically the Kullback-Leibler divergence.

This has very different upsides and downsides to the other methods. The result of VI is an analytic distribution of the form we chose. This can be very useful for sampling and analysis. However, if we chose a distribution which cannot fit the posterior very well, then we might get poor results. For example, optimizing the parameters of a single multivariate Gaussian can never cause it to become multi-modal, so if the posterior is multi-modal VI on a multivariate Gaussian could not fit this well.

The parameter optimization in VI is typically using a gradient descent-based optimizer, such as in Pyro [2] (a PPL library for Python, discussed more in Sections 3.1 & 3.3). This requires that we can also calculate the gradient of the parameters with respect to the loss function. Like with HMC, requiring the gradient has consequences, discussed in Section 4.1.

```python
from pyro.primitives import sample
import torch
import numpy as np
import pyro
from pyro import sample
from pyro.infer import NUTS, MCMC
from pyro.distributions import Normal
from matplotlib import pyplot as plt

def bad():
    x = sample('x', pyro.distributions.Normal(0, 1))
    if x > 0:
        y = sample('y', Normal(10, 2))
    else:
        y = sample('y', pyro.distributions.Gamma(3, 3))

    return y

nuts_kernel = NUTS(bad)

mcmc = MCMC(nuts_kernel,
    num_samples=10000,
    warmup_steps=10
)
mcmc.run()
mcmc.summary()
samples = mcmc.get_samples()

print(samples)

fig, ax = plt.subplots()
ax.hist(np.array(samples["y"]), bins=50)
plt.show()
```

Listing 2. The second example from [16] (Figure 2a), implemented in Pyro. Use of the same variable name for different distribution types in different branches gives silently incorrect results

2.2. Variable Identifiers

A subtle implementation challenge when implementing Markov chain Monte Carlo (MCMC) methods is how to correctly associate variables between two different runs (traces) of the program. MCMC methods record program traces, and reuse the values of variables to prevent re-sampling and to explore efficiently. In order to reuse the variables values, we need to correctly the same variable from two traces of the program. Using the variable name alone is not sufficient, for example we can have the variable be sampled from two different kinds of distribution in different branches of an *if* statement, in which case we should not reuse its value.

As mentioned before in Section 2.1.4, there are a few methods to do this correctly, such as deriving the dependency structure of the program, and resampling all dependents. The more information we can re-use from the previous sample, the better the performance. This is discussed more in [12].

However, some current PPLs do not use these methods - instead they leave the variable association to the programmer. Notably, Stan allows using the same variable with different distribution types in different branches, and silently gives incorrect results when doing so, which is shown in [16].

In Listing 2 I show that the same is also true for Pyro, by re-implementing the same example that Stan fails on. Pyro uses user-provided names to associate samples between runs, but does not check that the names are of the same distribution kind in different branches. The resulting histogram is shown in Figure 2. The peak on the left is too high when compared to the correct result from [16].

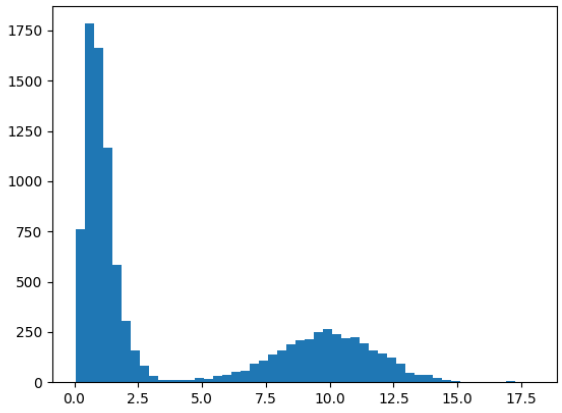

Fig. 2. Histogram of the posterior over y from Listing 2.

In Listing 3 Pyro correctly exits with an error that there are multiple variables with the name x within a single run.

3. COMPARISON

There are a lot of active probabilistic programming languages. Each has a history and motivation behind its existence. I discuss some of the notable ones below, then discuss some of the important differences, such as performance between them, in an effort to find the current Pareto front of PPL research.

I first present a quick overview then focus on the key differences.

3.1. Overview of existing PPLs

BUGS (*Bayesian Inference using Gibbs Sampling*) [17], was one of the first PPLs (sometimes called instead a PPS (Probabilistic Programming System)). BUGS only supports continuous variables and does not support user defined functions [18].

Stan [1] is one of the most popular PPLs, used widely in statistics. Stan introduced the Markov chain Monte Carlo method called NUTS (No U-Turn Sampling) [13]. Stan is very performant, because it translates models into C++, which are then compiled into native code. I will discuss this more in Section 3.4.

PyMC3 [2] is a library for Python, supporting gradient based MCMC methods and variational inference.

Pyro [2] is a fluent library for Python open sourced by Uber, which uses PyTorch or JAX to provide automatic differ-

```
from pyro.primitives import sample
import torch
import numpy as np
import pyro
from pyro import sample
from pyro.infer import NUTS, MCMC
from pyro.distributions import Normal
from matplotlib import pyplot as plt

def bad():
    x = sample('x', pyro.distributions.Normal(0, 1))
    for i in range(10):
        x = sample('x', pyro.distributions.Normal(x, 3))
    x

nuts_kernel = NUTS(bad)

mcmc = MCMC(nuts_kernel,
    num_samples=10,
    warmup_steps=10
)
mcmc.run()
mcmc.summary()
samples = mcmc.get_samples()

print(samples.keys())

fig, ax = plt.subplots()
ax.hist(np.array(samples["x"]), bins=50)
plt.show()
```

Listing 3. The third example from [16] (Figure 3a), implemented in Pyro. Pyro correctly exits with an error.

entiation. Pyro focuses strongly on the fact that it can represent any computable probability distribution, and is therefore a "Universal" PPL.

WebPPL [19] is a JavaScript-like PPL which can be run and embedded in a web browser. WebPPL supports most of the standard inference methods including variational inference.

BLOG [20] is a hybrid between probabilistic and logic programming languages, allowing "open-world" modeling. It uses MCMC methods for inference.

Anglican [21] aims to distill recent PPL research into a practical tool for the Clojure ecosystem.

Venture [4] is a higher-order PPL which aims to allow correctly combining different exact and approximate inference methods.

Turing.jl [3] is a fluent library for the Julia programming language. Turing.jl is to Julia as Pyro is to Python.

ThisPPL (as described by [7] but implemented by me) is a lisp-like language focused on conciseness and simplicity of implementing the language itself. It supports fewer inference algorithms.

3.2. Key differences

From the above brief overview we can see there are a lot of ways the languages differ:

- Platform. WebPPL targets web browsers while most are native applications.
- Paradigm. BLOG supports logic programming.
- Inference Algorithms. Older languages BUGS and Stan support fewer algorithms.
- Library vs. Standalone languages. Pyro, Turing, Anglican and WebPPL are libraries for their host languages Python, Julia, Clojure and JavaScript respectively - the remainder are standalone languages.
- Execution model. Most of these languages are interpreted or hosted in a JIT compiled language, but Stan models are translated to C++ code and compiled to native binaries.

In the next section I compare three different languages. I have chosen languages that differ on the above properties.

3.3. Comparison on Example Programs

The three languages I chose to compare are:

- Stan, because it is widely used, and it is compiled rather than interpreted.
- Pyro, because it is a library rather than a standalone PPL, and supports advanced inference via automatic differentiation.
- ThisPPL, because it is a standalone interpreted language (and I am interested to see how my language compares).

I compare their support for two models, how concise the code is on each model, as well as informally comparing their usability based on my personal experience implementing these models. Usability of programming languages is commonly compared with frameworks such as the cognitive dimensions framework [22]. However, such comparisons typically use user studies which is not feasible for this essay.

Six code listings are found in Appendix A. For each of these programming languages, I implemented both the linear regression model from above , and the classic "sprinkler" model.

3.3.1. Linear regression

The linear regression model (Listings 4, 5 & 6) tests support for inference on continuous distributions and being a very simple model is a good test for how concise a language is.

All three languages support this model, however we can see some noticeable differences.

On this program Stan is the most concise of the three due to implicit vector maths and implicit observation statements (note that the observation values come from a separate file not here pictured).

Pyro is almost as concise, except for the *import* statements. However, Pyro has a notable problem which is that

the names of variables must be repeated twice – once for the python variable, and once for the Pyro runtime.

ThisPPL is also relatively concise, not counting note the expanded loop which is done for clarity in the example on page 2.

3.3.2. *Sprinkler*

The sprinkler model (Listings 7, 8 & 9) is a classic example from introductions to Bayesian probability. It has binary variables only, and we easily can compute the posterior exactly for any observation. Here I am using it to test support for inference on discrete variables, and to test support for programs with control flow. The sprinkler model is as follows

- The weather is cloudy half the time.
- The weather is 80% to be raining if it is cloudy, or 20% likely otherwise.
- The sprinkler is 10% likely to be on if it is cloudy, and 50% likely otherwise.
- The grass is 99% likely to be wet if both the sprinkler is on and the weather is raining; 90% likely to be wet if only one of the sprinkler or raining; and 10% likely otherwise.

Stan does not support this model! In particular, Stan does not support latent discrete parameters, here the boolean variables *cloudy*, *sprinkler_on* and *raining*. When attempting to run the code pictured, we get the error "Parameters cannot be integers". There is no neat way around this - the Stan docs encourage the user to do algebra to marginalize out the discrete parameters [23]. Stan does not support these because it does not support computing the gradient through the discrete parameters - but note that Pyro can now do this, using the automatic differentiation from PyTorch.

Pyro and ThisPPL both do support this model. However note that Pyro only accepts tensors as random variables, which provides limitations to the flexibility of the programs. Pyro does support discrete variables, but booleans must be represented as integers. It also does not support compound objects or heterogeneous arrays as random variables.

3.4. Performance

I compared the performance of Pyro and Stan performing many samples on the linear regression program. The results are shown in Table 1.

Pyro and Stan are dramatically different in sampling speed on this program, as shown in Table 1. This is a factor of nearly 1000x, which even more significant than we would expect from the difference between Python and compiled binary.

In addition performing my own benchmarks on Stan and Pyro, I refer to the Uber AI paper "Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro" [24]. In this paper they benchmark Stan against Pyro and a new, JAX-based [25] implementation of Pyro.

Language	Model	# Samples	Total Time
Stan	Linear Regression	10000 samples	0.215 s
Pyro	Linear Regression	10000 samples	158 s

Table 1. Performance comparison between Pyro and Stan, both using NUTS MCMC with 10000 samples, 1 sample thinning and 1000 sample warmup.

Language	Model	Time per step
Stan (64-bit CPU)	HMM	0.53 ms
Pyro (32-bit CPU)	HMM	30.51 ms
NumPyro (32-bit CPU)	HMM	0.09 ms

Table 2. Benchmarks from [24]. Time (ms) per leapfrog step in different frame-works.

The results in Table 2 agree with my results that Pyro is significantly slower than Stan – however NumPyro has a mostly compatible API [5] and is faster than Stan, at least on this model.

4. TRADE-OFFS

In my discussion of inference algorithms and programming languages I identified some trade-offs that PPLs have made in the past. The first of these is whether to restrict programs to those where the gradient can be computed.

4.1. Gradient-based vs. non-gradient based

HMC and NUTS are the most efficient MCMC approximation methods. However, they require the gradient of the program to be computed. Stan can compute the gradient of all of the continuous distributions that it supports, and use the chain rule to calculate the gradient of the HMC dynamics with respect to the parameter values. However, using this method does not allow propagating the gradient through discrete variables such as booleans or categoricals.

This restricts the kinds of models that Stan can represent to models with continous latent variables.

However, automatic differentiation in programs such as Tensorflow, PyTorch and Julia now support differentiation through discrete latent variables using automatic marginalization. This has performance considerations, but unlocks the ability to use HMC on a larger variety of programs.

This is what the most recent generation of probabilistic programming languages have integrated. This feature is available in Pyro, Turing.jl and others.

This is progress, but still a trade-off, because automatic marginalization over discrete latent variables is slow, especially if there are many variables or they have many states.

Venture [4], from the MIT Probabilistic Computing Project, is attempting to solve this trade-off in a different way. They are working on allowing combinations of inference methods within one program. This is an interesting future research direction on this trade-off.

4.2. Standalone PPLs vs. libraries

Another trade-off in PPLs has historically been between standalone compiled languages like Stan and language-hosted DSL libraries such as PyMC [25].

As discussed in the language comparison, Stan models, including observations/data are translated to C++ and then compiled into native binaries. This allows a high level of optimization to high-performing native code, especially when compared to an interpreted language like Python. For this reason, Stan had been the most performant option until recently.

However, as a result of running Stan models as native binaries, it is more difficult to integrate the model with existing code. Although, in the case of Stan, this effort has been done. For example, the RStan interface to Stan can call the Stan binary to translate the model to C++, call the C++ compiler to compile the model, then collect and read the data into R after execution has finished.

On the other hand, libraries like PyMC have full integration with the host language. Apart from reading and writing data into a program, this also allows for assembling models dynamically, or calling arbitrary code during the sampling process, etc.

Recently however, this trade-off has again been lessened, notably by NumPyro [5] and Turing.jl [3]. These PPLs use aggressive JIT compilation to generate native code for a model before running it, while still being integrated into the host language.

Turing.jl uses Julia itself for the JIT compilation. I haven't been able to find citable benchmarks but there is discussion that Turing.jl is faster than Stan on some models, and in the ballpark on most. However, Turing.jl is very easily integrated with other parts of Julia.

NumPyro uses XLA [26] via JAX. XLA is a tracing JIT compiler for numeric computation programs which can be hosted in languages such as Python which do not themselves have a JIT compiler. As seen in Table 2, this can give performance comparable to or better than Stan.

With these libraries, the trade-off between standalone languages and libraries has fallen in favour of libraries, which now have as good performance as standalone languages while being better integrated with existing code.

5. FUTURE WORK IN PPLS

Throughout this report I identified 3 areas where future work in PPLs is promising.

The first is in the small section about variable identifiers. Here I discuss how Stan and Pyro give incorrect results because of incorrect reuse of values between the MCMC steps. Future work could be to discover and fix these errors in existing languages.

The second is the ongoing work by Venture to allow use of multiple inference algorithms in one program. This has the potential to greatly improve the performance / flexibility trade-off.

Lastly, because of the various restrictions of the different inference algorithms, and the trade-offs mentioned above, PPLs currently cannot be used effectively without understanding the inference algorithms. This unfortunately means that PPLs do not fulfill their goal of making inference available to users while requiring less knowledge. Users who do not understand the inference algorithms will find errors when they attempt to mix discrete and continuous variables, or bugs such as the one with variable identifiers. Future work here can be to improve usability by making errors helpful, and by acknowledging that users do need to learn about the inference algorithms.

6. SUMMARY: SHOULD YOU BE USING PPLS?

As discussed in Limitations (Section 1.4), Bayesian inference, and therefore PPLs, is limited to problems where the space of the unknown parameters is relatively small.

However, if you find yourself needing to solve a problem which has a small space of unknown parameters with Bayesian inference, a PPL is your best option for solving it. In particular, a recent PPL library such as NumPyro for Python or Turing.jl for Julia will give you both the best flexibility and very good performance.

PPLs are useful tools in their niche, and you might find them useful one day.

7. REFERENCES

[1] B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. A. Brubaker, J. Guo, P. Li, and A. Riddell, "Stan: A probabilistic programming language.," *Grantee Submission*, vol. 76, no. 1, pp. 1–32, 2017.

[2] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman, "Pyro: Deep Universal Probabilistic Programming," *Journal of Machine Learning Research*, 2018.

[3] K. Xu, H. Ge, and contributors., *Turing.jl*, https://github.com/TuringLang/Turing.jl, [accessed: 16-June-2021], 2021.

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

[5] *Getting started with NumPyro*, [accessed 21-June-2021], 2021. [Online]. Available: http://num.pyro.ai/en/stable/getting_started.html.

[6] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[7] J.-W. van de Meent, B. Paige, H. Yang, and F. Wood, *An Introduction to Probabilistic Programming*, https://arxiv.org/abs/1809.10756, 2018. arXiv: 1809.10756 [stat.ML].

[8] V. Dose, "Bayesian inference in physics: Case studies," *Reports on Progress in Physics*, vol. 66, no. 9, pp. 1421–1461, Aug. 2003. DOI: 10.1088/0034-4885/66/9/202. [Online]. Available: https://doi.org/10.1088/0034-4885/66/9/202.

[9] A. M. Ellison, "Bayesian inference in ecology," *Ecology letters*, vol. 7, no. 6, pp. 509–520, 2004.

[10] S. Chib and E. Greenberg, "Understanding the metropolis-hastings algorithm," *The american statistician*, vol. 49, no. 4, pp. 327–335, 1995.

[11] Wikipedia contributors, *Metropolis–hastings algorithm*, [accessed 21-June-2021], 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Metropolis%E2%80%93Hastings_algorithm&oldid=1027650047.

[12] S. Castellan and H. Paquet, "Probabilistic programming inference via intensional semantics," in *European Symposium on Programming*, Springer, 2019, pp. 322–349. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-17184-1_12.

[13] M. D. Hoffman and A. Gelman, "The no-u-turn sampler: Adaptively setting path lengths in hamiltonian monte carlo.," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1593–1623, 2014.

[14] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, vol. 112, no. 518, pp. 859–877, 2017. DOI: 10.1080/01621459.2017.1285773.

[15] Wikipedia contributors, *Variational bayesian methods*, [accessed 21-June-2021], 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Variational_Bayesian_methods&oldid=1025315825.

[16] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel, "A Provably Correct Sampler for Probabilistic Programs," in *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)*, P. Harsha and G. Ramalingam, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 45, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 475–488, ISBN: 978-3-939897-97-2. DOI: 10.4230/LIPIcs.FSTTCS.2015.475. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2015/5655.

[17] D. Spiegelhalter, A. Thomas, N. Best, and W. Gilks, "Bugs 0.5: Bayesian inference using gibbs sampling manual (version ii)," *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, pp. 1–59, 1996.

[18] A. Pfeffer, *Practical probabilistic programming*. Manning Publ., 2016.

[19] N. D. Goodman and A. Stuhlmüller, *The Design and Implementation of Probabilistic Programming Languages*, http://dippl.org, [accessed: 16-June-2021], 2014.

[20] B. Milch, B. Marthi, S. Russell, D. Sontag, D. L. Ong, and A. Kolobov, "1 blog: Probabilistic models with unknown objects," *Statistical relational learning*, p. 373, 2007.

[21] F. Wood, J. W. Meent, and V. Mansinghka, "A new approach to probabilistic programming inference," in *Artificial Intelligence and Statistics*, PMLR, 2014, pp. 1024–1032.

[22] T. R. G. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[23] S. contributors., *7 Latent Discrete Parameters*, [accessed 21-June-2021], 2021. [Online]. Available: https://mc-stan.org/docs/2_27/stan-users-guide/latent-discrete-chapter.html.

[24] D. Phan, N. Pradhan, and M. Jankowiak, "Composable effects for flexible and accelerated probabilistic programming in numpyro," *arXiv preprint arXiv:1912.11554*, 2019.

[25] F. C. Salvatier J Wiecki TV, *Probabilistic programming in Python using PyMC3*, https://peerj.com/articles/cs-55/, 2016.

[26] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing," *Systems for Machine Learning*, 2018.

8. APPENDIX A: CODE LISTINGS

8.1. Bayesian Linear Regression Model

```
data {
    int<lower=0> N;
    vector[N] x;
    vector[N] y;
}
parameters {
    real slope;
    real bias;
}
model {
    slope ~ normal(0, 5);
    bias ~ normal(0, 5);
    y ~ normal(bias + slope * x, 1);
}
```

Listing 4. Linear Regression in Stan

```
from pyro.primitives import sample
import torch
import numpy as np
from pyro import sample
from pyro.infer import NUTS, MCMC
from pyro.distributions import Normal
from matplotlib import pyplot as plt

def linear():
    x_data, y_data = [1, 2, 3, 4, 5, 6],
        torch.tensor([2.2, 4.2, 5.5, 8.3, 9.9, 12.1])
    slope = sample('slope', Normal(0, 5))
    bias = sample('bias', Normal(0, 5))

    for i in range(len(x_data)):
        x = x_data[i]
        mu = x * slope + bias
        y = sample(f"y_{i}", Normal(mu, 1), obs=y_data[i])

nuts_kernel = NUTS(linear)

mcmc = MCMC(nuts_kernel,
    num_samples=1000,
    warmup_steps=10
)
mcmc.run()
mcmc.summary()
samples = mcmc.get_samples()

print(samples)
```

Listing 5. Linear Regression in Pyro

```
(defn observe-data [x y_obs slope bias]
    (let [  y_pred (+ (* slope x) bias)
            sigma 1.0
            likelihood (normal y_pred sigma)]
        (observe likelihood y_obs)))

(let [  slope (sample (normal 0.0 10.0))
        bias  (sample (normal 0.0 10.0))
        xdata [ 1.0
                2.0
                3.0
                4.0
                5.0
                6.0 ]
        ydata [ 2.1
                3.9
                5.3
                7.7
                10.2
                12.9 ]]

    (observe-data (get xdata 0) (get ydata 0) slope bias)
    (observe-data (get xdata 1) (get ydata 1) slope bias)
    (observe-data (get xdata 2) (get ydata 2) slope bias)
    (observe-data (get xdata 3) (get ydata 3) slope bias)
    (observe-data (get xdata 4) (get ydata 4) slope bias)
    (observe-data (get xdata 5) (get ydata 5) slope bias)

    [slope bias])
```

Listing 6. Linear Regression in ThisPPL

8.2. Sprinkler Model

```
data {
   int<lower=0, upper=1> sprinkler_on;
   int<lower=0, upper=1> grass_wet;
}
parameters {
   int<lower=0, upper=1> cloudy;
   int<lower=0, upper=1> raining;
}
model {
    // It's cloudy half the time.
    cloudy ~ bernoulli(0.5);

    if (cloudy) {
        // If it's cloudy it's usually raining.
        raining ~ bernoulli(0.8);
    } else {
        // Sometimes it rains when it's not cloudy.
        raining ~ bernoulli(0.2);
    }

    if (cloudy) {
        // We don't usually turn the sprinkler on when
             it's cloudy
        sprinkler_on ~ bernoulli(0.1);
    } else {
        // We turn it on half of the time when it's fine.
        sprinkler_on ~ bernoulli(0.5);
    }

    if (sprinkler_on && raining) {
        // The grass is almost certainly wet if it's
             raining and the sprinkler is on.
        grass_wet ~ bernoulli(0.99);
    } else if (sprinkler_on || raining) {
        // the sprinkler and the rain each have a 90%
             chance of making the grass wet
        grass_wet ~ bernoulli(0.9);
    } else {
        // if neither the sprinkler nor the rain are
             going then the grass is almost certainly dry.
        grass_wet ~ bernoulli(0.01);
    }
}
```

Listing 7. Sprinkler in Stan. Note, this does not compile,

because Stan does not support discrete latent variables.

```
import torch
import pyro
from matplotlib import pyplot as plt

torch_true = torch.ones(())
torch_false = torch.zeros(())

def weather():
    # It's cloudy half the time.
    cloudy = pyro.sample('cloudy',
         pyro.distributions.Bernoulli(0.5))

    if cloudy:
        # If it's cloudy it's usually raining.
        raining = pyro.sample('raining',
             pyro.distributions.Bernoulli(0.8))
    else:
        # Sometimes it rains when it's not cloudy.
        raining = pyro.sample('raining',
             pyro.distributions.Bernoulli(0.2))

    if cloudy:
        # We don't usually turn the sprinkler on when
             it's cloudy
        sprinkler_on = pyro.sample('sprinkler_on',
             pyro.distributions.Bernoulli(0.1))
    else:
        # We turn it on half of the time when it's fine.
        sprinkler_on = pyro.sample('sprinkler_on',
             pyro.distributions.Bernoulli(0.5))

    if sprinkler_on and raining:
        # The grass is almost certainly wet if it's
             raining and the sprinkler is on.
        grass_wet_theta = 0.99
    elif sprinkler_on or raining:
        # the sprinkler and the rain each have a 90%
             chance of making the grass wet
        grass_wet_theta = 0.9
    else:
        # if neither the sprinkler nor the rain are going
             then the grass is almost certainly dry.
        grass_wet_theta = 0.01
    grass_wet = pyro.sample('grass_wet',
         pyro.distributions.Bernoulli(grass_wet_theta))

    return torch.tensor([cloudy, raining, sprinkler_on,
         grass_wet])

def conditioned_weather():
    pyro.condition(weather, data={"grass_wet":
         torch_true})

# nuts_kernel = pyro.infer.NUTS(conditioned_weather)

# mcmc = pyro.infer.MCMC(nuts_kernel,
#     num_samples=10000,
# )
# mcmc.run(weather)
# samples = mcmc.get_samples()
# print(samples)

importance = pyro.infer.Importance(conditioned_weather,
     guide=None, num_samples=1000)
thing = importance.run()
print(thing.get_normalized_weights())

def plot_bool_hist(ax, data, weights):
    data = np.array(data)
    ax.set_xlabel("Value")
    ax.set_ylabel("Mass")

    ax.set_xticks([0.5, 1.5])
    ax.set_xticklabels(["false", "true"])

    ax.hist(data, density=True, weights=weights, bins=2)
```

Listing 8. Sprinkler in Pyro

```
(let [
    cloudy (sample (flip 0.5))
    raining (sample (if cloudy
                        (flip 0.8)
                        (flip 0.2)))
    sprinkler_on (sample (if cloudy
                        (flip 0.1)
                        (flip 0.5)))
    grass_wet (sample (if (and sprinkler_on raining)
                        (flip 0.99)
                        (if (or sprinkler_on raining)
                            (flip 0.9)
                            (flip 0.01))))
    ]
    (observe grass_wet true)
    [cloudy raining sprinkler_on grass_wet])
```

Listing 9. Sprinkler in ThisPPL