LF-PPL: A Low-Level First Order Probabilistic Programming Language for Non-Differentiable Models

Yuan Zhou*,1 Bradley J. Gram-Hansen*,1 Tobias Kohn2,† Tom Rainforth1 Hongseok Yang3 Frank Wood4
1University of Oxford 2University of Cambridge 3KAIST 4University of British Columbia

Abstract

We develop a new Low-level, First-order Probabilistic Programming Language (LF-PPL) suited for models containing a mix of continuous, discrete, and/or piecewise-continuous variables. The key success of this language and its compilation scheme is in its ability to automatically distinguish parameters the density function is discontinuous with respect to, while further providing runtime checks for boundary crossings. This enables the introduction of new inference engines that are able to exploit gradient information, while remaining efficient for models which are not everywhere differentiable. We demonstrate this ability by incorporating a discontinuous Hamiltonian Monte Carlo (DHMC) inference engine that is able to deliver automated and efficient inference for non-differentiable models. Our system is backed up by a mathematical formalism that ensures that any model expressed in this language has a density with measure zero discontinuities to maintain the validity of the inference engine.

1 Introduction

Non-differentiable densities arise in a huge variety of common probabilistic models [1, 2]. Often, but not exclusively, they occur due to the presence of discrete variables. In the context of probabilistic programming [3, 4, 5, 6] such densities are often induced via branching, i.e. if-else statements, where the predicates depend upon the latent variables of the model. Unfortunately, performing efficient and scalable inference in models with non-differentiable densities is difficult and algorithms adapted for such problems typically require specific knowledge about the discontinuities [7, 8, 9], such as which variables the target density is discontinuous with respect to and catching occurrences of the sampler crossing a discontinuity boundary. However, detecting when discontinuities occur is difficult and problem dependent. Consequently, automating specialized inference algorithms in probabilistic programming languages (PPLs) is challenging.

To address this problem, we introduce a new Low-level First-order Probabilistic Programming Language (LF-PPL), with a novel accompanying compilation scheme. Our language is based around carefully chosen mathematical constraints, such that the set of discontinuities in the density function of any model written in LF-PPL will have measure zero. This is an essential property for many inference algorithms designed for non-differentiable densities [7, 8, 9, 10, 11]. Our accompanying compilation scheme automatically classifies discontinuous and continuous random variables for any model specified in our language. Moreover, this scheme can be used to detect transitions across discontinuity boundaries at runtime, providing important information for running such inference schemes.

Relative to previous languages, LF-PPL enables one to incorporate a broader class of specialized inference techniques as automated inference engines. In doing so, it removes the burden from the user of manually establishing which variables the target is not differentiable with respect to. Its low-level nature is driven by a desire to establish the minimum language requirements to support inference engines tailored to problems with measure-zero discontinuities, and to allow for a formal proof of correctness. Though still usable in its own right, our main intention is that it will be used as a compilation target for existing systems, or as an intermediate system for designing new languages.

There are a number of different derivative-based inference paradigms for which LF-PPL can help extend to non-differentiable setups [7, 8, 9, 10, 11]. Of particular note, are stochastic variational inference...
Different methods have been suggested to improve inference performance in models with discontinuous densities. For example, they use sophisticated integrators in the HMC setting to remain effective when there are discontinuities. Analogously, in the variational inference and deep learning literature, reparameterization methods have been proposed that allow training for discontinuous targets and discrete variables [9, 20].

However, these advanced methods are, in general, not incorporated in existing gradient-based PPSs, as existing systems do not have adequate support to deal with the discontinuities in the density functions of the model defined by probabilistic programs. This is usually necessary to guarantee the correct execution of those inference methods in an automated fashion, as many require the set of discontinuities to be of measure zero. That is, the union of all points where the density is discontinuous have zero measure with respect to the Lebesgue measure. In addition to this, some further methods require knowledge of where the discontinuities are, or at least catching occurrences of discontinuity boundaries being crossed.

Of particular relevance to our language and compilation scheme are compilers which compile the program to an artifact representing a direct acyclic graphical model (DAG), such as those employed in BUGS [27] and, in particular, the first order PPL (FOPPL) explored in [28]. Although the dependency structures of the programs in our language are established in a similar manner, unlike these setups, programs in our language will not always correspond to a DAG, due to different restrictions on our density factors, as will be explained in the next section. We also impose necessary constraints on the language by limiting the functions allowed to ensure that the advanced inference processes remain valid.

3 The Language

LF-PPL adopts a Lisp-like syntax, like that of Church [4] and Anglican [5]. The syntax contains two key special forms, sample and observe, between which the distribution of the query is defined and whose interpretation and syntax is analogous to that of Anglican.

More precisely, sample is used for drawing random variables, returning that variable, and observe factors the density of the program using existing variables and fixed observations, returning zero. Both special forms are designed to take a distribution object as input, with observe further taking an observed value. These distribution objects form the elementary random variables of the language and are constructed using one of a number of internal constructors for common objects such as normal and bernoulli. Figure 1 shows an example of an LF-PPL program.
(let [x (sample (uniform 0 1))]
  (if (< (- q x) 0)
    (observe (normal 1 1) y)
    (observe (normal 0 1) y))
  (< (- q x) 0))

Figure 1: An example LF-PPL program sampling x from a uniform random variable and invoking a choice between two observe statements that factor the trace weight using different Gaussian likelihoods. The (< (- q x) 0) term, which is usually written as ((q – x) < 0), represents a Bernoulli variable parameterized by q and its boolean value also corresponds to which branch of the if statement is taken. The slightly unusual writing of the program is due to its deliberate low-level nature, with almost all syntactic sugar removed. One sugar that has been left in for exposition is an additional term in the let block, i.e. (let [x e] e e), which can be trivially unraveled.

A distribution object constructor of particular note is factor, which can only be used with observe. Including the statement (observe (factor log-p) _) will factor the program density using the value of (exp log-p), with no dependency on the observed value itself (here _). The significance of factor is that it allows the specification of arbitrary unnormalized target distributions, quantified as log-p which can be generated internally in the program, and thus have the form of any deterministic function of the variables that can be written in the language.

Unlike many first-order PPLs, such as that of [28], LF-PPL programs do not permit interpretation as DAGs because we allow the observation of internally sampled variables and the use of factor. This increases the range of models that can be encoded and is, for example, critical in allowing undirected models to be written. LF-PPL programs need not correspond to a correctly normalized joint formed by the combination of prior and likelihood terms. Instead we interpret the density of a program in the manner outlined by [29] §4.3.2 and §4.4.3, noting that for any LF-PPL program, the number of sample and observe statements (i.e. \( n_x \) and \( n_y \) in their notation) must be fixed, a restriction that is checked during the compilation.

To formalize the syntax of LF-PPL, let us use x for a real-valued variable, c for a real number, op for an analytic primitive operation on reals, such as +, –, *, / and exp, and d for a distribution object whose density is defined with respect to a Lebesgue measure and is piecewise smooth under analytic partition (See Definition [1]). Then the syntax of expressions e in our language are given as:

\[
e ::= x | c | (op e ... e) | (if (< e 0) e e) | (let [x e] e)
  | (sample (d e ... e)) | (observe (d e ... e) c)
\]

Our syntax is deliberately low-level to permit theoretical analysis and aid the exposition of the compiler. However, common syntactic sugar such as for-loops and higher-level branching statements can be trivially included using straightforward unravellings. Similarly, we can permit discrete variable distribution objects by noting that these can themselves be desugared to a combination of continuous random variables and branching statements. Thus, it is straightforward to extend this minimalistic framework to a more user-friendly language using standard compilation approaches, such that LF-PPL will form an intermediate representation. For implementation and code, see https://github.com/bradleygramhansen/PyLFPPPL

4 Compilation Scheme

We now provide a high-level description of how the compilation process works. Specifically, we will show how it transforms an arbitrary LF-PPL program to a representation that can be exploited by an inference engine that makes of use of discontinuity information.

The compilation scheme performs three core tasks: a) finding the variables which the target is discontinuous with respect to, b) extracting the density of the program to a convenient form that can be used by an inference engine, and c) allowing boundary crossings to be detected at runtime. Key to providing these features is the construction of an internal representation of the program that specifies the dependency structure of the variables, the Linearized Intermediate Representation (LIR). The LIR contains vertices, arcs, and information of the if predicates. Each vertex of the LIR denotes a sample or observe statement, of which only a finite and fix number can occur in LF-PPL. The arcs of the LIR define both the probabilistic and condition dependencies of the variables. The former of these are constructed in same was as is done in the FOPPL compiler detailed in [28].

Using the dependency structure represented by the LIR, we can establish which variables are capable of changing the path taken by a program trace, that is the change the branch taken by one or more if statements. Because discontinuities only occur in LF-PPL through if statements, the target must be continuous with respect to any variables not capable of changing the traversed path. We can thus mark these variables as being “continuous”. Though it is possible for the target to still be continuous with respect to variables that appear in, or have dependent variables appearing in, the branching function of an if statement, such cases cannot, in general, be statically established. We therefore mark all such variables as “discontinuous”.

To extract the density to a convenient form for the inference engine, the compiler transforms the program into a collection four sets—\( \Delta, \Gamma, D, \) and \( F \)—by recur-
We finish the section by noting two limitations of the
A second limitation is that changes in the vector of
whether the current sample falls into the
true
condition for
1. the
2. the
3. N
4. \(M_i\), \(O_i\)
5. the

\[ G(x) = \sum_{i=1}^{N} \left( \prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0] \cdot h_i(x) \right) \]

where
1. the \(p_{i,j}, q_{i,l} : \mathbb{R}^k \to \mathbb{R}\) are analytic;
2. the \(h_i : \mathbb{R}^k \to \mathbb{R}\) are smooth;
3. \(N\) is a positive integer or \(\infty\);
4. \(M_i, O_i\) are non-negative integers; and
5. the indicator functions

\[
\prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0]
\]

for the indices \(i\) define a partition of \(\mathbb{R}^k\), that is, the following family forms a partition of \(\mathbb{R}^k\):

\[
\left\{ \left\{ x \in \mathbb{R}^k \mid \forall j \ p_{i,j}(x) \geq 0, \forall l \ q_{i,l}(x) < 0 \right\} \right\} 1 \leq i \leq N.
\]

Intuitively, \(G\) is a function defined by partitioning \(\mathbb{R}^k\) into finitely or countably many regions and using a smooth function \(h_i\) within region \(i\). The products of the indicator functions of these summands form a partition of \(\mathbb{R}^k\), so that only one of these products gets
evaluated to a non-zero value at \( x \). To evaluate the sum, we just need to evaluate these products at \( x \) one-by-one until we find one that returns a non-zero value. Then, we have to compute the function \( h_i \) corresponding to this product at the input \( x \). Even though the number of summands (regions) \( N \) in the definition is countably infinite, we can still compute the sum at a given \( x \).

**Theorem 1.** If an unnormalized density \( \mathcal{P} : \mathbb{R}^n \rightarrow \mathbb{R}_+ \) has the form of Definition 1 and so is piecewise smooth under analytic partition, then there exists a (Borel) measurable subset \( A \subseteq \mathbb{R}^n \) such that \( \mathcal{P} \) is differentiable outside of \( A \) and the Lebesgue measure of \( A \) is zero.

The proof is given in Appendix A. The target density being almost everywhere differentiable with discontinuities of measure zero is an important property required by many inference techniques for non-differentiable models. As we shall prove in Section 5.2 any program that can be compiled in LF-PPL constructs a density in the form of Definition 1 and thus satisfies this necessary condition.

5.2 Translation Rules

5.2.1 Overview

The compilation scheme \( e \rightsquigarrow (\Delta, \Gamma, D, F) \) translates a program, which can be denoted as an expression \( e \) according to the syntax in Section 3, to a quadruple of sets \((\Delta, \Gamma, D, F)\). The first set \( \Delta \) represents the set of all sampled random variables. All variables generated from `sample` statements in \( e \) will be recognized and stored in \( \Delta \). Variables that have not occurred in any `if` predicate are guaranteed to be continuous. Otherwise, they will be also put in \( \Gamma \subseteq \Delta \), as the overall density is discontinuous with respect to them. \( D \) represents the densities from `sample` statements and has the form of a set of the pairs, i.e., \( D = \{ (\eta_1, k_1), \ldots, (\eta_N, k_N) \} \), where \( N_D \) is the number of the pairs, \( \eta \) denotes a product of indicator functions indicating the partition of the space, and \( k \) represents the products of the densities defined by the `sample` statements. The last set \( F \) contains the densities from `observe` statements and the return expression of \( e \). It is a set of tuples \( F = \{ (\zeta_1, l_1, v_1), \ldots, (\zeta_{N_F}, l_{N_F}, v_{N_F}) \} \), where \( N_F \) is the number of the tuples, \( \zeta \) functions similar to \( \eta \), \( l \) is the product of the densities defined by `observe` statements and \( v \) denotes the returning expression. Note that it is a design choice to have \( v \) included in \( F \).

Given \( e \rightsquigarrow (\Delta, \Gamma, D, F) \), one can then construct the unnormalized density defined by the program \( e \) as

\[
\mathcal{P} := \left( \sum_{i=1}^{N_D} \eta_i \cdot k_i \right) \cdot \left( \sum_{j=1}^{N_F} \zeta_j \cdot l_j \right)
\]

which by Theorem 2 will be piecewise smooth under analytic partitions.

Recall that by assumption, the density of each distribution type \( d \) is piecewise smooth under analytic partition when viewed as a function of a sampled value and its parameters. Thus, we can assume that the probability density of a distribution has the form in Definition 1. For each distribution \( d \), we define a set of pairs \( \Phi(d) = \{(\psi_1, \phi_1), \ldots, (\psi_{N_d}, \phi_{N_d})\} \) where \( N_d \) is the number of the partitions, \( \psi \) denotes the product of indicator functions indicating the partition of the space, taking the form of \( \prod_{j=1}^{M} \mathbb{1}[p_{i,j}(x) \geq 0], \prod_{i=1}^{N} \mathbb{1}[q_{i,j}(x) < 0] \), and \( \phi \) represents a smooth probability density function within that partition. One can then construct the probability density function \( \mathcal{P}_d \) for \( d \) from \( \Phi(d) \). For given parameters \( x_1, \ldots, x_s \) of the distribution \( d \) and a given `sample` value \( x_0 \), we let \( x = (x_0, \ldots, x_s) \) and the probability density function defined by \( d \) is

\[
\mathcal{P}_d(x_0; x_1, \ldots, x_s) = \sum_{n=1}^{N_d} \psi_n(x) \cdot \phi_n(x)
\]

For example, given \( x_0 \) drawn from normal distribution \( \mathcal{N}(\mu, \sigma) \), we have \( \Phi(d) = \{(1, \mathcal{N}(\mu, \sigma))\} \) and \( \mathcal{P}_d(x_0; \mu, \sigma) = \mathcal{N}(x_0; \mu, \sigma) \). Similarly a uniform \( U(a, b) \) sampled variable \( x_0 \) has \( \Phi(d) \) as

\[
\{(1[x_0-a<0], 0), (1[b-x_0<0], 0), (1[x_0-a\geq0 \land b-x_0\geq0], U(x_0; a, b))\}
\]

and \( \mathcal{P}_d = \{1[x_0-a\geq0 \land b-x_0\geq0] \cdot U(x_0; a, b)\} \). Note that in practice one can omit the pair \((\psi_n, \phi_n)\) in \( \Phi(d) \) when \( \phi_n = 0 \) for simplicity and the probability density in the region denoting by the corresponding \( \psi_n \) is zero.

5.2.2 Formal Translation Rules

The translation process \( e \rightsquigarrow (\Delta, \Gamma, D, F) \), is defined recursively on the structure of \( e \). We present this recursive definition using the following notation

\[
\begin{align*}
\text{premise} & \quad \text{conclusion} \\
\end{align*}
\]

which says that if the premise holds, then the conclusion holds too. Also, for real-valued functions \( f(x_1, \ldots, x_n) \) and \( f'(x_1, \ldots, x_n) \) on real-valued inputs, we write \( f[x_i := f'] \) to denote the composition \( f(x_1, \ldots, x_{i-1}, f'(x_1, \ldots, x_n), x_{i+1}, \ldots, x_n) \). We now define the formal translation rules.

The first two rules define how we map the set of variables \( x \) and the set of constants \( c \) to their unnormalized density and the values at which they are evaluated.

\[
\begin{align*}
\text{For inputs} & \quad \text{output} \\
\{x\} & \rightsquigarrow \{(1,1), \{1,1,x\}\} \\
\{c\} & \rightsquigarrow \{(\emptyset, \emptyset), \{(1,1), \{(1,1,c)\}\}\}
\end{align*}
\]

The third rule allows one to translate the primitive operations \( \text{op} \) defined in the LF-PPL, such as +, -, * and / with their argument expressions \( c_1 \) to \( c_n \), where \( c_1 \) to \( c_n \) will be evaluated first. Note that \((\eta_i, k_i) \in D_i \) represents the enumeration of all \((\eta_i, k_i)\) pairs in \( D_i \) and the result of this operation among all the \( D_i \) is the
The fourth rule for control flow operation if enables us to translate the predicate \((< e_1 0)\), its consequent \(e_2\) and alternative \(e_3\). This provides us with the semantics to correctly construct a piecewise smooth function, that can be evaluated at each of the partitions.

\[
e_i \sim (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, 2, 3
\]

\[
D' = \{\{\prod_{i=1}^n \eta_i, \prod_{i=1}^n k_i\} \mid (\eta_i, k_i) \in D_i\}
\]

\[
F' = \{\{(\zeta_i \cdot \zeta_2 \cdot l_i \cdot l_2, \psi, \phi) \mid (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
\text{(if } (< e_1 0) \text{ if } e_2 \text{ else } e_3 \text{) } \sim (\bigcup_{i=1}^n \Delta_i, \bigcup_{i=1}^n \Gamma_i, D', F')
\]

The translation rule for the sample statement generates a random variable from a specific distribution. During translation, we pick a fresh variable, i.e. a variable with a unique name to represent this random variable and add it to the \(\Delta\) set. Then we compose the density of this variable according to the distribution \(d\) and corresponding parameters \(e_i\).

\[
e_i \sim (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, \ldots, n
\]

pick a fresh variable \(z\)

\[
\Delta' = \{\{\psi \cdot \prod_{i=1}^n \zeta_i, \phi \cdot \mathbf{x} := (z, v_1, \ldots, v_n)\} \mid (\psi, \phi) \in \Phi(d), (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
D' = \{\{\prod_{i=1}^n \eta_i, \prod_{i=1}^n \kappa_i\} \mid (\eta_i, \kappa_i) \in D_i\}
\]

\[
F' = \{\{(\psi \cdot \prod_{i=1}^n \zeta_i, \phi \cdot \mathbf{x} := (y, v_1, \ldots, v_n)\} \mid (\psi, \phi) \in \Phi(d), (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
\text{(sample } (d e_1, \ldots, e_n) \text{) } \sim (\Delta', \Gamma', D', F')
\]

The translation rule for the observe statement, different from the sample expression, factors the density according to the distribution object, with all parameters \(e_i\) and the observed data \(c\) evaluated.

\[
e_i \sim (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, \ldots, n
\]

\[
\Delta' = \{\{\psi \cdot \prod_{i=1}^n \zeta_i, \phi \cdot \mathbf{x} := (x, v_1, \ldots, v_n)\} \mid (\psi, \phi) \in \Phi(d), (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
D' = \{\{\prod_{i=1}^n \eta_i, \prod_{i=1}^n \kappa_i\} \mid (\eta_i, \kappa_i) \in D_i\}
\]

\[
F' = \{\{(\psi \cdot \prod_{i=1}^n \zeta_i, \phi \cdot \mathbf{x} := (c, v_1, \ldots, v_n)\} \mid (\psi, \phi) \in \Phi(d), (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
\text{(observe } (d e_1, \ldots, e_n) \text{) } \sim (\Delta', \Gamma', D', F')
\]

The translation rule for let expressions first translates the definition \(e_1\) of \(x\) and the body \(e_2\) of let, and then joins the results of these translations. When joining the \(\Delta\) and \(\Gamma\) sets, the rule checks whether \(x\) appears in the sets from the translation of \(e_2\), and if so, it replaces \(x\) by variable names appearing in \(e_1\), an expression that defines \(x\). Although let is defined as single binding, we can construct the rules to translate the let expression, defining and binding multiple variables by properly desugaring.

\[
e_i \sim (\Delta_i, \Gamma_i, D_i, F_i) \text{ for } i = 1, 2
\]

\[
\Delta_0 = \{\{\zeta_i, l_i, v_i\} \in F_i \text{ and } z \text{ occurs free in } v_1\}
\]

\[
\Delta' = \Delta_1 \cup (\Delta_2 \setminus \{x\}) \cup (\{x \in \Delta_2\text{ where } \Delta_0 \text{ else } 0\}
\]

\[
\Gamma' = \Gamma_1 \cup \Gamma_2 \setminus \{x\} \cup (\{x \in \Gamma_2\text{ where } \Delta_0 \text{ else } 0\}
\]

\[
D' = \{\{(\zeta_1 \cdot \eta_2[x := v_1], k_1 \cdot k_2[x := v_1]) \mid (\eta_1, k_1) \in D_i, (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
F' = \{\{(\zeta_1 \cdot \eta_2[x := v_1], l_1 \cdot l_2[x := v_1], v_2[x := v_1]) \mid (\zeta_i, l_i, v_i) \in F_i\}
\]

\[
\text{(let } x e_1 \text{) } e_2 \sim (\Delta', \Gamma', D', F')
\]

Theorem 2. If \(e\) is an expression that does not contain any free variables and \(e \sim (\Delta, \Gamma, D, F)\), then the unnormalized density defined by \(e\) is in the form of Equation [7]. It is a real-valued function on the variables in \(\Delta\), which is non-negative and piecewise smooth under analytic partition as per Definition [4].

The proof is provided in Appendix [3]. By providing this set of mathematical translations we have been able to prove that any such program written in LF-PPL constructs a density in the form of Definition [4] which is piecewise smooth under analytic partitions. Together with Theorem [4] we further show that this density is almost everywhere differentiable and the discontinuities are of measure zero, a necessary condition for several inference schemes such as DHMC [5].

5.3 A Compilation Example

We now present a simple example of how the compiler transforms the program \(e_{pp}\) in Figure [4] to the quadruple \((\Delta_{pp}, \Gamma_{pp}, D_{pp}, F_{pp})\). The translation rules are applied recursively and within each rule, all individual components are compiled eagerly first. Namely, we step into each individual component and step out until it is fully compiled. A desugared version of \(e_{pp}\) is:

\[
\text{(let } [x \text{ sample (uniform 0 1))]}
\]

\[
\text{let } x_\_ \text{ (if } (< (- q x) 0) \text{)}
\]

\[
\text{observe (normal 1 1) y})
\]

\[
\text{observe (normal 0 1) y})
\]

\[
(< (- q x) 0))
\]

where \(q\) and \(y\) are constant and \(x_\_\) is not used. It follows the following steps.

i. Rule (let \(x e_1\text{,out} e_2\text{,out})\). We start by looking at the outer let expressions, with \(e_1\text{,out}\) being the sample statement and \(e_2\text{,out}\) corresponding to the entire inner let block. Before we can generate the output of this rule, we step into \(e_1\text{,out}\) and \(e_2\text{,out}\) and compile them accordingly.

ii. Rule (sample \(d e_1 e_2))\). We then apply the sample rule on \(e_1\text{,out} := \text{(sample (uniform 0 1))}\)
from \( \mathbb{I} \) with each of its components evaluated first. For \( \text{let } (0, 1) \) and we have \( 0 \sim (0, 0, \{(1, 1), \{(1, 1, 0)\}) \) and \( 1 \sim (0, 0, \{(1, 1), \{(1, 1, 1)\})) \), \( d \) represents \text{uniform} distribution and has the form 

\[
\Phi(d) = \{(\mathbb{I}[x \geq 0], \mathbb{I}[1-x \geq 0], \mathcal{U}(;0,1))\}.
\]

After combining each set following the rule, with a fresh variable \( z \), we have \( e_1, out \sim (\{z\}, 0, \{(1, 1), \{(1, 1, (q-x))\}) \) with \( (q-x) \) as an operation - applied to \( q \) and \( x \).

\( e_2 \) and \( e_3 \) both follow \text{observe} \( (d, e_2, c) \). Take \( e_2 := (\text{observe} \text{ normal } 1 1 y) \) as an example, 1 is constant and \( d \) is the \text{normal} distribution and has \( \Phi(d) = \{(1, \mathcal{N}(y; 1, 1))\} \). We combine each set and have \( e_2 \sim (0, 0, \{(1, 1), \{(1, \mathcal{N}(y; 1, 1), 0)\}) \). Similarly, \( e_3 \sim (0, 0, \{(1, 1), \{(1, \mathcal{N}(y; 0, 1), 0)\}) \).

With \( e_1, e_2 \) and \( e_3 \) all evaluated, we can now continue the \text{if} rule. The key features are to extract variables in \( e_1 \) and take the densities on each branch respectively. As a result, \( e_1, in \) compiles to \( \Delta = \{x\}, \Gamma = \{x\}, \{\{1, 1\}\} \) and \( F = \{(\mathbb{I}[q-x < 0], \mathcal{N}(y; 1, 1), 0), (\mathbb{I}[q-x \geq 0], \mathcal{N}(y; 0, 1), 0)\} \).

\( \text{Result of the inner let.} \) Together with the outcome from \( \text{let} \) and \( \text{if} \) we can continue compiling the inner \text{let} block as in \( \text{let} \) and it is translated to 

\[
\Delta = \{x\}, \Gamma = \{x\},
\]

\[
D = \{(\mathbb{I}[q-x < 0], 1), (\mathbb{I}[q-x \geq 0], 1)\}
\]

\[
F = \{(\mathbb{I}[q-x < 0], \mathcal{N}(y; 1, 1), (q-x < 0)), \}
\]

\[
(\mathbb{I}[q-x \geq 0], \mathcal{N}(y; 0, 1), (q-x < 0))\}
\]

\( \text{Result of the outer let.} \) Finally, with \( e_1, out \) compiled in \( \text{let} \) and \( e_2, out \) in \( \text{let} \) we step out to \( \text{let} \). It is worth to emphasize that the variables \( \Delta \) are the sampled ones rather than what are named in the \text{let} expression, i.e. \( x \) and \( x_\Gamma \). Here \( x \) is replaced by \( z \) as declared in \( e_1, out \) by following the \text{let} rule, and we have the final quadruple output:

\[
\Delta_{pp} = \{z\}, \Gamma_{pp} = \{z\},
\]

\[
\begin{align*}
D_{pp} &= \{(\mathbb{I}[z \geq 0], \mathbb{I}[1-z \geq 0], \mathbb{I}[q-z < 0], \mathcal{U}(z; 0, 1)),
\end{align*}
\]

\[
\begin{align*}
F_{pp} &= \{(\mathbb{I}[q-z < 0], \mathcal{N}(y; 1, 1), (q-z < 0)), \}
\end{align*}
\]

From the quadruple, we have the overall density as 

\[
\mathcal{P} = \mathcal{I}[x \geq 0], \mathcal{I}[1-x \geq 0], \mathcal{I}[q-x < 0], \mathcal{U}(z; 0, 1), \mathcal{N}(y; 1, 1) + \mathcal{I}[z \geq 0], \mathcal{I}[1-z \geq 0], \mathcal{I}[q-z < 0], \mathcal{U}(z; 0, 1), \mathcal{N}(y; 0, 1), \]

We can also detect when any random variable in \( \Gamma \), in this case \( z \), has crossed the discontinuity, by checking the boolean value of the predicate of the \text{if} statement \( \langle -q \times 0 \rangle \).
We compared the Mean Squared Error (MSE) of We construct the GMM as follows: which represents a hyperbolic-like potential function, with MwG approach. The results are shown in Figure 2 within-Gibbs (MwG) in PyMC3 [22], with the same problem increases. We consider the following density[7] as the number of clusters and N the total number of data. The Categorical distribution is constructed by a combination of uniform draws and nested if expressions, as shown in Appendix [D] For our experiments, we considered a simple case with $\mu_0 = 0$, $\sigma_0 = 2$, $\sigma_{z_{1:N}} = 1$ and $p_0 = [0.5, 0.5]$, along with the synthetic dataset: $y_{1:N} = [-2.0, -2.5, -1.7, -1.9, -2.2, 1.5, 2.2, 3, 1.2, 2.8]$. We compared the Mean Squared Error (MSE) of the posterior estimates for the cluster means of both an unoptimized version of DHMC and an optimized implementation of NUTS with Metropolis-within-Gibbs (MwG) in PyMC3 [22], with the same computation budget. We take $10^5$ samples and discard $10^4$ for burn in. We find that our DHMC implementation, performs comparable to the NUTS with MwG approach. The results are shown in Figure 2 as a function of the number of samples.

### 6.2 Heavy Tail Piecewise Model

In our next example we show how the efficiency of DHMC improves, relative to vanilla HMC, on discontinuous target distributions as the dimensionality of the problem increases. We consider the following density[7] which represents a hyperbolic-like potential function,

$$\pi(x) = \begin{cases} 
\exp(-\sqrt{x^T A x}) & \text{if } \|x\|_{\infty} \leq 3 \\
\exp(-\sqrt{x^T A x} - 1) & \text{if } 3 < \|x\|_{\infty} \leq 6 \\
0 & \text{otherwise}
\end{cases}$$

It generates planes of discontinuities along the boundaries defined by the if expressions. To write this as a density in our language we make use of the factor distribution object as shown in Appendix [D]

The results in Figure 3 provide a comparison between the DHMC and the standard HMC on the worst mean absolute error [7] as a function of the number of iterations and time, WMAE($N$) = $\frac{1}{N} \max_{\text{dimensions}} \left[ \sum_{n=1}^{N} x_{d,n}^{(n)} \right]$. We see that as the dimensionality of the model increases, the per-sample performance of HMC deteriorates rapidly as seen in the top row of Figure 3 Even though DHMC is more expensive per iteration than HMC due to its sequential nature, in higher dimensions, the additional time costs occurred by DHMC is much less than the rate at which HMC performance deteriorates. The reason for this is that the acceptance rate of the HMC sampler degrades with increasing dimension, while the coordinate-wise integrator of the DHMC sampler circumvents this.

### 7 Conclusion

In this paper we have introduced a Low-level First-order Probabilistic Programming Language (LF-PPL) and an accompanying compilation scheme for programs that have non-differentiable densities. We have theoretically verified the language semantics via a series of translations rules. This ensures programs that compile in our language contain only discontinuities that are of measure zero. Therefore, our language together with the compilation scheme can be used in conjunction with other scalable inference algorithms such as adapted versions of HMC and SVI for non-differentiable densities, as we have demonstrated with one such variant of HMC called discontinuous HMC. It provides a roadmap for incorporating other inference algorithms into PPSs and shows the performance improvement of these inference algorithms over existing ones.
Acknowledgements

Yuan Zhou is sponsored by China Scholarship Council (CSC). Bradley Gram-Hansen is supported by the UK EPSRC CDT in Autonomous Intelligent Machines and Systems (CDT in AIMS). 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. Yang was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177). Kohn and Wood were supported by DARPA D3M, Intel as part of the NERSC Big Data Center, and NSERC under its Discovery grant and accelerator programs.

References


A Proof of Theorem 1

Proof. Assume that $\mathcal{P}$ is piecewise smooth under analytic partition. Thus,

$$\mathcal{P}(x) = \sum_{i=1}^{N} \prod_{j=1}^{M_i} \mathbb{1}[p_{i,j}(x) \geq 0] \cdot \prod_{l=1}^{O_i} \mathbb{1}[q_{i,l}(x) < 0] \cdot h_i(x)$$

for some $N, M_i, O_i$ and $p_{i,j}, q_{i,l}, h_i$ that satisfy the properties in Definition 1.

We use one well-known fact: the zero set $\{x \in \mathbb{R}^n \mid p(x) = 0\}$ of an analytic function $p$ is the entire $\mathbb{R}^n$ or has zero Lebesgue measure [30]. We apply the fact to each $p_{i,j}$ and deduce that the zero set of $p_{i,j}$ is $\mathbb{R}^n$ or has measure zero. Note that if the zero set of $p_{i,j}$ is the entire $\mathbb{R}^n$, the indicator function $\mathbb{1}[p_{i,j} \geq 0]$ becomes the constant-1 function, so that it can be omitted from the RHS of equation (2). In the rest of the proof, we assume that this simplification is already done so that the zero set of $p_{i,j}$ has measure zero for every $i, j$.

For every $1 \leq i \leq N$, we decompose the $i$-th region

$$R_i = \{ x \mid p_{i,j} \geq 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l \}$$

(3)

to

$$R'_i = \{ x \mid p_{i,j} > 0 \text{ and } q_{i,l}(x) < 0 \text{ for all } j, l \}$$

$$R''_i = R_i \setminus R'_i.$$ (4)

Note that $R'_i$ is open because the $p_{i,j}$ and $q_{i,l}$ are analytic and so continuous, both $\{r \in \mathbb{R} \mid r > 0\}$ and $\{r \in \mathbb{R} \mid r < 0\}$ are open, and the inverse images of open sets by continuous functions are open. This means that for each $x \in R'_i$, we can find an open ball at $x$ inside $R_i$ so that $\mathcal{P}(x') = h_i(x')$ for all $x'$ in the ball. Since $h_i$ is smooth, this implies that $\mathcal{P}$ is differentiable at every $x \in R'_i$.

For the other part $R''_i$, we notice that

$$R''_i \subseteq \bigcup_{j=1}^{M_i} \{ x \mid p_{i,j}(x) = 0 \}.$$ The RHS of this equation is a finite union of measure-zero sets, so it has measure zero. Thus, $R''_i$ also has measure zero as well.

Since $\{R_i\}_{1 \leq i \leq N}$ is a partition of $\mathbb{R}^n$, we have that

$$\mathbb{R}^n = \bigcup_{i=1}^{N} R'_i \cup \bigcup_{i=1}^{N} R''_i.$$ The density $\mathcal{P}$ is differentiable on the union of $R'_i$’s. Also, since the union of finitely or countably many measure-zero sets has measure zero, the union of $R''_i$’s has measure zero. Thus, we can set the set $A$ required in the theorem to be this second union. □

B Proof of Theorem 2

Proof. As shown in Equation 1

$$\mathcal{P} := \left( \sum_{i=1}^{N_D} \eta_i \cdot k_i \right) \cdot \left( \sum_{j=1}^{N_F} \zeta_j \cdot l_j \right)$$

it suffices to show that both factors are non-negative and piecewise smooth under analytic partition, because such functions are closed under multiplication.

We prove a more general result. For any expression $e$, let Free($e$) be the set of its free variables. Also, if a function $\mathcal{G}$ in Definition 1 satisfies additionally that its $h_i$’s are analytic, we say that this function $\mathcal{G}$ is piecewise analytic under analytic partition. We claim that for all expressions $e$ (which may contain free variables), if $e \rightsquigarrow (\Delta, \Gamma, D, F)$, where $D = \{(\eta_i, k_i)\}_{1 \leq i \leq N_D}$ and $F = \{(\zeta_j, l_j)\}_{1 \leq j \leq N_F}$, then $\left( \sum_{i=1}^{N_D} \eta_i \cdot k_i \right)$ and $\left( \sum_{j=1}^{N_F} \zeta_j \cdot l_j \right)$ are non-negative functions on variables in Free($e$) $\cup \Delta$ and they are piecewise analytic under analytic partition, as $k$ and $l'$ in the sum are analytic. These two properties in turn imply that $\left( \sum_{i=1}^{N_D} \eta_i \cdot k_i \right) \cdot \left( \sum_{j=1}^{N_F} \zeta_j \cdot l_j \right)$ is a function on variables in Free($e$) $\cup \Delta$ and it is also piecewise analytic (and thus piecewise smooth) under analytic partition. Thus, the desired conclusion follows. Regarding our claim, we can prove it by induction on the structure of the expression $e$. □
C Discontinuous Hamiltonian Monte Carlo

The discontinuous HMC (DHMC) algorithm was proposed by [8]. It uses a coordinate-wise integrator, Algorithm 1, coupled with a Laplacian momentum to perform inference in models with non-differentiable densities. The algorithm works because the Laplacian momentum ensures that all discontinuous parameters move in steps of \( \pm m_b \epsilon \) for fixed constants \( m_b \) and step size \( \epsilon \), where the index \( b \) is associated to each discontinuous coordinate. These properties are advantages because they remove the need to know where the discontinuity boundaries between each region are; the change of the potential energy in the state before and after the \( \pm m_b \epsilon \) move provides us with information of whether we have enough kinetic energy to move into this new region. If we do not have enough energy we reflect backwards \( p_b = -p_b \). Otherwise, we move to this new region with a proposed coordinate update \( x^*_b \) and momentum \( p_b - m_b \cdot \text{sign}(p_b) \cdot \Delta U \). This is in contrast to algorithms such as Reflect, Refract HMC [7], that explicitly need to know where the discontinuities boundaries are. Hence, it is important to have a compilation scheme that enables one to do that.

The addition of the random permutation \( \phi \) of indices \( b \) is to ensure that the coordinate-wise integrator satisfies the criterion of reversibility in the Hamiltonian. Although the integrator does not reproduce the exact solution, it nonetheless preserves the Hamiltonian exactly, even if the density is discontinuous. See Lemma 1 and Theorems 2-3 in [8]. This yields a rejection-free proposal.

Algorithm 1 Coordinate-wise Integrator. A random permutation \( \phi \) on \( \{1, \ldots, B\} \) is appropriate if the induced random sequences \( \phi(1), \ldots, \phi(|B|) \) and \( \phi(|B|), \ldots, \phi(1) \) have the same distribution

\begin{algorithm}
\begin{algorithmic}[1]
\Function{Coordinatewise}{x, p, \epsilon, U}
\State \text{pick an appropriate random permutation } \phi \text{ on } B
\For{i = 1, \ldots, B}
\State b \leftarrow \phi(i)
\State x^* \leftarrow x
\State x^*_b \leftarrow x^*_b + \epsilon m_b \cdot \text{sign}(p_b)
\State \Delta U \leftarrow U(x^*) - U(x)
\If{K(p_b) = m_b \cdot |p_b| > \Delta U}
\State x_b \leftarrow x^*_b
\State p_b \leftarrow p_b - m_b \cdot \text{sign}(p_b) \cdot \Delta U
\Else
\State p_b \leftarrow -p_b
\EndIf
\EndFor
\State \Return x_b, p_b
\EndFunction
\end{algorithmic}
\end{algorithm}

Then DHMC algorithm [8] adapted for LF-PPL and our compilation scheme is as follows:
Algorithm 2 Discontinuous HMC Integrator for the LF-PPL.

χ is a map from random-variable names \( n \) in \( \Delta \) to their values \( x_n \), \( H \) is the total Hamiltonian, \( \epsilon > 0 \) is the step size, and \( L \) is the trajectory length.

1: function DHMC-LFPPL(\( \Delta, \Gamma, D, F, x, p, H, \epsilon, L \))
2: \( B = \Gamma; \quad A = \Delta \setminus \Gamma \)
3: for \( a \in A \) do \( \triangleright a \) represents the set of continuous variables
4: \( x_a^0 \leftarrow x_a; \quad p_a \sim \mathcal{N}(0, 1) \)
5: end for
6: for \( b \in B \) do \( \triangleright b \) represents the set of discontinuous variables
7: \( x_b^0 \leftarrow x_b; \quad p_b \sim \text{Laplace}(0, 1) \)
8: end for
9: \( \forall a \in A, \ x_a^0 \leftarrow x_a; \quad p_a \sim \mathcal{N}(0, 1) \) \( \triangleright A \) represents the set of continuous variables
10: \( \forall b \in B, \ x_b^0 \leftarrow x_b; \quad p_b \sim \text{Laplace}(0, 1) \) \( \triangleright B \) represents the set of discontinuous variables
11: \( U \leftarrow -\text{LogJointDensity}(D, F) \)
12: for \( i = 1 \) to \( L \) do
13: \( U_A \leftarrow U \) with names in \( B \) replaced by their values in \( x_B^i \)
14: \( (x_A^i, p_A^i) \leftarrow \text{HALFSTEP1}(x_A^{i-1}, p_A^{i-1}, \epsilon, U_A) \)
15: \( U_B \leftarrow U \) with names in \( A \) replaced by their values in \( x_A^i \)
16: \( (x_B^i, p_B^i) \leftarrow \text{COORDINATE-WISE}(x_B^{i-1}, p_B^{i-1}, \epsilon, U_B) \)
17: \( U_A \leftarrow U \) with names in \( B \) replaced by their values in \( x_B^i \)
18: \( (x_A^i, p_A^i) \leftarrow \text{HALFSTEP2}(x_A^i, p_A^i, \epsilon, U_A) \)
19: end for
20: \( x^L \leftarrow x_A^L \cup x_B^L, \ p^L \leftarrow p_A^L \cup p_B^L \)
21: \( x^*, p^* \leftarrow \text{EVALUATE}(F, x^L, p^L) \)
22: \( \alpha \sim \text{Uniform}(0, 1) \)
23: if \( \alpha > \min\{1, \exp(H(x, p) - H(x^*, p^*))\} \) then
24: \( \quad \text{return } x^*, p^* \)
25: else
26: \( \quad \text{return } x, p \)
27: end if
28: end function
29: function HALFSTEP1(\( x, p, \epsilon, U \))
30: \( p' \leftarrow p - \frac{\epsilon}{2} \nabla_x U(x) \)
31: \( x' \leftarrow x + \frac{\epsilon}{2} \nabla_p K(p') \)
32: return \( (x', p') \)
33: end function
34: function HALFSTEP2(\( x, p, \epsilon, U \))
35: \( x' \leftarrow x + \frac{\epsilon}{2} \nabla_p K(p) \)
36: \( p' \leftarrow p - \frac{\epsilon}{2} \nabla_x U(x') \)
37: return \( (x', p') \)
38: end function
D Program code

(let [y (vector -2.0 -2.5 ... 2.8)
      pi [0.5 0.5]
      z1 (sample (categorical pi))
      ...
      z10(sample (categorical pi))
      mu1 (sample (normal 0 2))
      mu2 (sample (normal 0 2))
      mus (vector mu1 mu2)]
  (if (< (- z1) 0)
    (observe (normal mu1 1) (nth y 0))
    (observe (normal mu2 1) (nth y 0))))

Figure 4: The LF-PPL version of the Gaussian mixture model detailed in Section 6.

(let [x (sample (uniform -6 6))
      abs-x (max x (- x))
      z (- (sqrt (* x (* A x))))]
  (if (< (- abs-x 3) 0)
    (observe (factor z) 0)
    (observe (factor (- z 1)) 0))
  x)

Figure 5: The LF-PPL version of the heavy-tailed model detailed in Section 6.