Usability of Probabilistic Programming Languages

Alan F. Blackwell
University of Cambridge

Tobias Kohn
University of Cambridge

Martin Erwig
Oregon State University

Atilim Güneş Baydin
University of Oxford

Luke Church
Africa's Voices Foundation

James Geddes
The Alan Turing Institute

Andy Gordon
Microsoft Research

Maria Gorinova
University of Edinburgh

Bradley Gram-Hansen
University of Oxford

Neil Lawrence
Amazon AI Research

Vikash K. Mansinghka
MIT

Brooks Paige
The Alan Turing Institute

Tomas Petricek
University of Kent

Diana Robinson
University of Cambridge

Advait Sarkar
Microsoft Research

Oliver Strickson
The Alan Turing Institute

Abstract

This discussion paper presents a conversation between researchers having active interests in the usability of probabilistic programming languages (PPLs), but coming from a wide range of technical and research perspectives. Although PPL development is currently a vigorous and active research field, there has been very little attention to date to basic questions in the psychology of programming. Relevant issues include mental models associated with Bayesian probability, end-user applications of PPLs, the potential for data-first interaction styles, visualisation or explanation of model structure and solver behaviour, and many others.

Introduction

This discussion has been convened to consider open research questions, priorities and potential design guidelines relevant to the usability of probabilistic programming languages (PPLs). It provides a short introduction, for the PPIG audience, to the conceptual and operational principles that underlie PPLs. It then discusses two alternative perspectives: firstly an applications perspective, in which there might be potential for broader application of PPL methods in the context of end-user programming if the languages were usable by a wider range of people; and secondly an educational perspective, in which we consider whether PPLs might be a valuable tool for teaching principles of probability, or even as an introduction to programming that takes a fundamentally probabilistic rather than deterministic or imperative view of how computation should be conceived. Finally, an agenda is suggested for a "furthest-first" approach to education and applications.

Background and History

Probabilistic programming is a paradigm, generally embedded within conventional languages, in which programmers can define random variables and perform probabilistic inference using high-level language constructs. Note that the phrase 'probabilistic programming' does *not* refer to the application of probabilistic methods in program synthesis, or in programming by example, which are also topics of interest at PPIG, but are not the subject of this paper. Typical random variables might be a sample data set, observable system output, model parameters, or other latent variables, usually understood within a framework of generative modeling and Bayesian inference (Kim & Pearl 1983, Pearl & Mackenzie 2018). A key distinction between probabilistic and conventional programming languages is that random variables do not have a single value, but should be regarded as defining (or sampling from) a probability distribution of likely values. Performing inference amounts to finding probable values for some of these variables given the values of other variables, e.g. inferring the parameters of a model given observed data. Popular inference algorithms include Markov chain Monte Carlo (MCMC) (Wingate 2011) and variational inference (Blei 2017).

Probabilistic programming can have a ``declarative" flavor: probabilistic languages typically separate the executable code that defines a probabilistic model from the inference algorithm(s) that are actually invoked to find probable values. In this sense, probabilistic programming might be compared to embeddings of logic programming in conventional languages, in which declarative operations can be mixed with ordinary constructs for I/O, etc. However, expert probabilistic programmers may have to read a probabilistic model imperatively, using their understanding of the underlying inference algorithm to diagnose inference failures and to try to improve inference performance. This breaks the illusion of declarative behavior, analogously to the need for expert Prolog programmers to understand backtracking search. Additional technical discussion about distinctions between familiar declarative languages and the generative nature of probabilistic programming is reported in an appendix.

There exist a number of different approaches to probabilistic programming that are built around a variety of semantics and inference engines. One important class of languages are those that only allow random choices in restricted positions, ensuring that programs are guaranteed to define a finite set of random variables. Stan (Gelman, Lee, and Guo 2015), BUGS (Spiegelhalter et al. 1996) and Infer.NET (Minka et al. 2013) are prominent examples with many real-world use cases. van de Meent et al. 2018 calls these languages "first-order" (FOPPLs). Another important class of languages are those that are computationally universal, allowing random choices in any statement or expression. Church (Goodman et al. 2008), Venture (Mansinghka, Selsam, Perov 2014), Anglican (Wood, van de Meent, and Mansinghka 2014), Pyro (Bingham et al. 2018), TensorFlow Probability (Dillon et al. 2017, Tran et al. 2017) and PyProb (Baydin et al. 2019) are all examples of this class. Another important class of languages are those that support *programmable inference,* i.e. they enable the user to customize inference using high-level constructs, to meet the performance and accuracy requirements of a given application . Examples include Venture (Mansinghka, Selsam, Perov 2014; Mansinghka et al. 2018), Pyro (Bingham et al. 2018), and Gen (Cusumano-Towner et al. 2019).

Research in this field has primarily been driven by the desire for effective tools to enable statistical and machine learning research, and there has been little specialist attention to studying the usability of PPLs, or designing features that enhance usability. There has also been relatively little attention to PPLs in the human-centric computing, software engineering, software visualisation or visual languages communities, with the exception of a small number of experiments conducted by authors of this paper (systems built by

authors Gorinova, Erwig and Gordon are discussed below, along with unpublished work in progress by authors Geddes, Strickson and Robinson).

The Idea of a Probabilistic Programming Language

The diversity of technical approaches just described mean that there is no single conception of what probabilistic programming provides. There is even less consensus on where PPL approaches might take us in the future (for example, when used as an implementation platform for experiments with deep generative models, or when probabilistic programs scale up to 100s or 1000s of lines of code). Nevertheless, it is useful to consider why this paradigm offers a distinctive intellectual appeal, in terms of the role of computation within a scientific enquiry. Let's consider one of the possible styles of probabilistic programming, in which we focus on simplicity, interpretability, and causality.

Our modern understanding of the world started with a revolutionary insight and discovery. While analysing the data of the position of stars and planets in the night skies, astronomer Johannes Kepler found a function that reliably describes the data, and hence models the movements of planets in the solar system. A simple function was able to capture the 'big' data, supported predictions, and led to new scientific insights. Finding a function that describes a given set of data has since appeared in different shapes and forms. Carl Gauss had a clear model of how the function should look like, but had to deal with imprecision and uncertainty in the data when he developed least-squares linear regression. And Joseph Fourier's description of data through trigonometric functions builds a basis of today's signal processing. In contrast to Kepler's great feat of finding a new model, subsequent methods have mostly focused on adapting a known (or assumed) model to the data.

In recent years, AI research has made significant impact with neural networks - a universal set of functions that can model a wide variety of data. With higher power computing systems, deep and complex networks can describe many data sets with surprising precision. However, there is a catch: in their universality, neural networks provide little insight about the actual underlying models behind the data. We often struggle to understand exactly how a neural network describes a given set of data: the price of universality.

Like Gauss and Fourier, however, we often do have an understanding of what the model behind the data should look like. For linear regression, for instance, we assume a linear relationship in the data set and could thus replace the potentially huge and complex model of a neural network with a much simpler model featuring just a few parameters. All we need is a method to find meaningful values for the parameters in our model, whatever model we choose. This is one of the valuable opportunities offered by probabilistic programming methods.

Probabilistic programming in this sense combines the descriptive power of simple models with sophisticated methods to adapt the parameters in your model to given data. At the core of probabilistic programming are a set of "inference algorithms" not unlike the "learning algorithms" one encounters when training a neural network. However, instead of training a universal neural network using data samples, the user writes a specific model in a probabilistic programming language and infers its parameters through conditioning on data. As with earlier generations of declarative logic programming and expert system knowledge representation languages, a probabilistic program is not run in the classical sense, but instead makes inferences. AI advocates sometimes say, of neural network optimization, that the system is being 'taught', rather than programmed - but to apply this analogy to probabilistic programming neglects the work done by the PPL programmer, and in particular the potential in some languages to implement alternative inference models (a layer of abstraction where the computation is expressed in more conventional imperative form, e.g. PLDI 2018).

How does statistical modelling relate to probability?

Classical linear regression is built on the principle of least squares: on the idea that there is a single pair of parameter values for which the "error" between model and data is minimal. However, in reality, linear regression hardly ever returns the true underlying parameters exactly - although we expect the result to be close to the true values, at least if the linear model is a good fit for the data. Our confidence in the proposed parameter values will then also increase when more data points are captured by the function. On the flipside, if the linear model is a bad fit for the data in the first place, the algorithm will never produce truly meaningful and accurate values.

In the context of probabilistic programming, we do not seek a single value for each parameter of the model, hoping that all of these values when taken together will make our overall model a good fit to the data. It is much more natural in the Bayesian context to think of the value of each parameter in terms of probability distributions. Instead of reporting the single 'best' (most likely) value for each parameter of the model, the inference engine can tell you how likely any value for that parameter would be - as a posterior distribution that is informed by your prior expectation regarding the model structure.

Think of it this way: if your model really fits your data, the inference engine will single out a range of values for your parameters that makes the entire model a very probable candidate for describing the data. If the model does not fit your data, the inference engine will find that no specific set of parameter values really stands out, and that no combination of values would make your model a particularly probable explanation for the data.

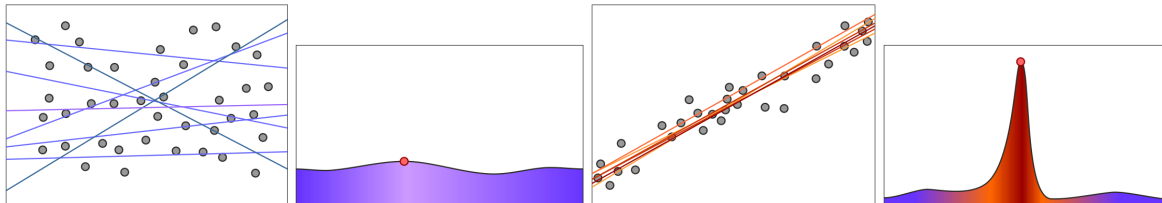

Figure 1: If the linear model is *not* a good fit for the data (on the left), no line really stands out, and all possible parameters have more or less the same probability (the probability density function has many peaks, each corresponding to a different line on the scatter plot, with little similarity between them and none clearly better). Even though we might find a "best" fit, it does not really distinguish itself from the rest. However, if the linear model is a good fit for the data (on the right, with a single clearly distinguished peak in the probability density function corresponding to many similar lines on the scatter plot), then some choices for parameter values are clearly better than others and stand out as highly probable values.

As noted in the introduction to this section, we have presented a relatively straightforward interpretation of probabilistic programming, to illustrate the potential appeal of the paradigm as a scientific tool and educational approach. Current and future developments in PPLs include far more complex approaches to modelling and generation, that might not necessarily retain this intuitive appeal. Nevertheless, this straightforward style of application offers a starting point for broadening access to PPL methods, as discussed in the following sections.

The end-user development perspective on PPLs

Several PPL research projects have aimed to make PPLs accessible to end-user programmers, especially spreadsheet users and database users. PPLs also present new usability challenges that fall especially heavily on end users who lack a deep understanding of the impact of approximation on their results.

How can we make probabilistic programming accessible to spreadsheet users?

It is often observed that more people create programs in spreadsheets such as Excel than in all other programming languages combined (Scaffidi et al 2005). Many business data processing applications that would have required professional programmers to implement them in the 1960s or 70s are now routinely created by people who have never received any formal training in programming, but are able to use spreadsheets to implement a wide variety of straightforward accountancy and data processing applications. The spreadsheet paradigm is approachable in part because of the way that it offers a concrete perspective on the object of interest (the user's data) rather than on the abstractions of programming. Nevertheless, it is possible to extend the spreadsheet paradigm with sophisticated abstract capabilities such as those of functional programming languages (Peyton Jones et al 2003).

Commercial extensions to the spreadsheet paradigm are generally driven by the business needs of spreadsheet users, who have practical problems to solve rather than being driven by technical curiosity or research agendas. These are defined as end-user programmers (Blackwell 2006, 2017), end-user developers (Lieberman et al 2006) or even end-user software engineers (Ko et al 2011). Increased business interest in the methods of statistical data science suggests that these end-users are likely to find value in PPL capabilities, especially if presented in an interaction context such as a spreadsheet, where users would be able to construct and interpret the behaviour of their program in the context of the data that it relates to. The usability advantages of data-centric presentation have already been observed in previous evolution of statistical applications, such as the adoption of a spreadsheet-style data view in the popular SPSS, when the SPSS-X release under Windows 2.0 included a tabular data editor that became established as the primary user interface for the GUI versions of the product.

Considering PPLs from this data-centric perspective, in terms of the tasks of end-users, raises interesting questions about the boundaries of the programming task. End-users who are creating simple scripts or macros to automate repetitive actions often deal implicitly with the attention investment tradeoff, calculating whether the programming effort will pay off in saved time (Blackwell 2002). Because spreadsheets allow exceptional conditions to be handled by direct manipulation (just changing the cells concerned), spreadsheet programmers are far less likely to devote a lot of effort to anticipating infrequent situations in their code. This is a very familiar situation to data scientists responsible for "data wrangling" - formatting, organising and cleaning data sets for statistical analysis. Although standard research and teaching data sets have been cleaned in advance, making wrangling distinct from modelling, real-world data science involves a far more ambiguous relationship between the two. It is often difficult to judge whether unexpected data values are errors, outliers, or important clues to an inappropriate model. If probabilistic programming involved closer interaction with original data, this would provide the opportunity for "wrangling" operations to inform the programmed model. It would also provide the opportunity for the mundane tasks of wrangling to be automated through inference, as in the Data Noodles prototype by some of the authors (Gorinova et al. 2016) that allows users to demonstrate how they would like their data to be arranged in a table, then searches for a set of structural transformations that will generate that table.

The same redefinition of boundaries, in a data-centric approach to end-user probabilistic programming, might allow us to revisit the definition of labelling, in the machine learning lifecycle. At present, most supervised learning systems rely on data that has been labelled with a “ground truth” of human interpretation, often obtained via Mechanical Turk, or forced tasks such as ReCAPTCHA. The people who carry out these labelling tasks may have some insight into the modelling assumptions (for example in relation to implicit bias in the judgments they have been asked to make, or explanations of why they made a particular judgment). However, those insights are often not captured, or may even be discarded, in conventional machine learning paradigms. There have been some experiments in semi-supervised or mixed-initiative approaches to labelling, for example supporting more dynamic structuring of label categories. However, more sophisticated approaches could be enabled if the data views presented to the labeller offered more direct insight into the structure and behaviour of the model, perhaps even allowing trusted labellers to make incremental adjustments or modifications to the structure. A complementary benefit would be realised by data scientists themselves, who are often advised to spend more time looking at the data, before making assumptions about the structure of the model. Allowing the end-user programmer to contribute to labelling in a way that was continuous with the modelling task allows more sophisticated reasoning across multiple levels of abstraction, in a manner that is analogous to the constant shifts in level of abstraction that are observed in studies of expert programmers (Pennington 1987, 1995)

End-user paradigms such as spreadsheet programming also demonstrate the advantages for learning that result from bridging across levels of abstraction. Modern user interfaces appear more intuitive because a handful of basic principles can be applied in a concrete manner, together with discoverability of more abstract functions and relations. All of these design principles could be applied to implement tools supporting methodologies such as the Bayesian workflow of explore, model, infer, check, repeat (Gabry et al 2019).

We can also consider the potential to generate spreadsheets from PPL specifications. Two of the authors of this paper (Geddes and Strickson) are working on a probabilistic programming language – nocell – where the result of running a program written in this language is a spreadsheet model applied to the input data. This allows the advantages of spreadsheet models, of understandability and immediacy, to be combined with sophisticated modelling techniques, as well as good software development practices (such as version control and modularity). A further aim is to connect the communities of software-developer data analysts with the wider community of spreadsheet users.

Values in nocell are probability distributions, supporting arithmetic operations and conditioning on observed data. This is motivated in part by the observation that many spreadsheet models are used in situations where capturing uncertainty in the model is beneficial, and that recent advances in PPL and machine language ideas could provide significant value to users of these models, who would otherwise have limited access to tools built on these ideas. This probabilistic approach contrasts with, for example, the type of scenario analysis that is commonly performed, where "typical", "typical-low", "typical-high" and perhaps more extreme values of model inputs are considered, to obtain an idea of the range of values that can be produced by a model (this could still be useful as a *presentational* tool).

In the nocell approach, the programmer constructs a model as a nocell program, which includes setting appropriate program inputs to probability distributions and perhaps describe observations of their values. When run, this program produces a spreadsheet where the program outputs are evaluated from particular choices of input value, but in addition are annotated with their mean and standard deviation.

An important consideration, driving some of the current work, is how the probability distribution of a value of interest should be represented within the spreadsheet. This should be done in a way that conveys useful information at a level of detail appropriate for a wide audience.

In fact, spreadsheets have served as a site for probabilistic programming almost since their inception, and there is a class of user, the *end-user probabilistic programmer* (Borghouts et al 2019) who does probabilistic programming in the spreadsheet. There are standard techniques for building Monte Carlo simulations in spreadsheets using the RAND() function. The commercial extensions @Risk and Crystal Ball automate the details of Monte Carlo inference and in so doing amount to probabilistic programming languages, albeit without Bayesian update. In his dissertation, Streit (2008) describes a spreadsheet that supports uncertainty types, such as a number explicitly tagged as an estimate, or a numeric interval, or a probability distribution. Streit argues that such uncertain values should propagate through spreadsheet calculations. Borghouts et al (2019) conjecture that at present, there may be as many as ten times more end-user probabilistic programmers in spreadsheets, than programmers who use mainstream PPLs.

Another promising area of application of PPLs is in the design of information systems that advise doctors when they are making medical decisions. Diagnostic reasoning is a natural fit for application of PPLs since doctors use Bayesian reasoning as they gather evidence from examining patients and conducting tests and update their confidence in particular diagnoses accordingly. One barrier to adoption of these languages for medical professionals is that they may not have the statistical or computing expertise. If PPLs could be adapted into a form that would be approachable to doctors, starting with what they are and the main advantages of using them, decision support systems could potentially be designed in a way that is helpful and supportive of their reasoning and approaches. Designing these tools with the help of doctors is key both to their usefulness as well as widespread adoption. Understanding the tools builds trust, which is essential to their uptake as doctors will need to first understand and then communicate and support their findings to their patients.

How can we make PPLs accessible to database users?

Database users, especially those familiar with SQL, are another important and widespread group of end-user programmers, To make PPLs accessible to these users, we need to address several usability challenges. First, database users are often accustomed to SQL-like languages and data types, and may be unfamiliar with the general-purpose programming languages in which PPLs are often embedded. We thus need PPLs that are tightly integrated into existing database environments, and that offer SQL-like interface for specifying models and performing inference. Second, database users are unlikely to be familiar with the mathematics needed to diagnose failures of approximate inference. We thus need to develop PPLs in which inference is guaranteed to be accurate, so that database users can understand system outputs. Third, database users are likely to have qualitative domain knowledge and access to data, but may not know how to encode domain knowledge in the form of a probabilistic model. We thus need ways to empower these users with automated ways of building models from data, and also provide them with automated methods for adjusting those programs in light of incremental changes to the database schema and source data.

Some PPLs have begun to explore ways to address these challenges. For example, the BayesDB platform offers an SQL-like probabilistic language for building models and using these models to infer missing and/or unlikely values in the data (Saad & Mansinghka 2016; Mansinghka et al. 2015). It also provides automated Bayesian data modeling mechanisms that build baseline models for tabular data (Mansinghka et al. 2009, 2016). These models are written in domain-specific probabilistic languages (Saad et al. 2019) in which inference can be implemented exactly (Saad & Mansinghka 2016), and learned from data via Monte Carlo methods that scale to databases with tens of millions of cells (Obermeyer et al. 2014). BayesDB can use this models for automated exploratory data analysis, e.g. to report Bayesian inferences about the probable structure and strength of predictive relationships between variables (Saad & Mansinghka, 2017).

The educational perspective on PPLs

Languages such as Scratch (Resnick et al 2009) include both an application domain (in the case of Scratch, an architecture for agent-based graphical canvas operations) and an educationally-oriented IDE (in the case of Scratch, a block-syntax editor and a library browser). The graphics application domain echoes the emphasis on graphics in many earlier educational languages from Logo to AgentSheets and Alice. Such languages bring a Piagetian perspective to computation, for example the Scratch sprite or Logo turtle help the learner to think *syntonically* about program execution, by allowing the learner to reason about a computational agent's behaviour (Watt 1998, Pane 2002). Furthermore, as often advocated by Kay, tangible representations can help provide a concrete manipulable representation that helps learners to reason about abstract relations (Repenning 1996, Edge 2006, Kohn 2019).

Beyond the cognitive and notational advantages of a graphical application domain, a core motivation has been that graphics are fun. Children enjoy drawing, and freedom of graphical expression can help bring a creative and exploratory attitude to the introduction of novel notation systems (Stead 2014). How do we make teaching about (Bayesian) probability fun, through use of an application domain that motivates learners? Much traditional teaching of probability follows the traditions of Bayes himself, and other early theorists, in exploring the mathematical implications of gambling (coin tosses, dice throws, card shuffling etc). Teaching of frequentist statistics is largely grounded in the logic of hypothesis testing, and taught in the service of biology or psychology. Might contemporary problems of data science be more motivational as an application domain for education? For example, local children are affected by traffic speed on a road outside their school. The council reports an average speed slightly under the speed limit as evidence that there is no danger. Would children be motivated by gaining access to the council's raw data, and exploring the implications of those distributions for themselves?

A probabilistic programming IDE might potentially include live visualisations of the model; direct dependencies and probability distributions, highlighting conditional independencies, as well as providing tools for visual or numerical diagnostics. Some of these ideas have been explored by previous work. For example, some of the present authors (Gorinova et al 2016) presented a live, multiple-representation environment (MRE) for the probabilistic programming language Infer.NET. Alongside the Infer.NET code, the environment maintains a visualisation of the program as a Bayesian network (a directed acyclic graph, encoding the conditional dependencies between variables). The marginal distribution of each variable is also visualised. Gorinova et al (2016) show that, when presented with debugging and program description tasks, users inexperienced in probabilistic modelling are faster and more confident when using the MRE compared to when using a conventional programming environment. Participants were also more likely to give a higher-level description of the dependencies in the model when using the MRE, as opposed to a lower-level code description. This suggests that live visualisations can be a useful way of

teaching core concepts in Bayesian reasoning, and of drawing a clear distinction between conventional and probabilistic programming.

At PPIG 2018, Andrea diSessa made the provocative suggestion that school science lessons such as physics should in future be taught through students constructing their own computational simulations of the phenomenon, rather than through algebraic analysis and fitting of experimental observations. Scientific simulation has been a common application domain for educational programming languages in the past, for example in Repenning's AgentSheets, and Cypher's KidSim. We might imagine the possibility that teaching of probability in schools could be better achieved through modelling in a PPL. Indeed, Goodman and Tenenbaum's *Probabilistic Models of Cognition* includes interactive code examples implemented in WebPPL, Lee and Wagenmakers text on *Bayesian Cognitive Modelling* uses BUGS, and Andrew Gelman's courses at Columbia such as *Statistics GR6103* use modelling in Stan. At school level, we can imagine that physical demonstration apparatus such as a Galton Board could be simulated in software (this suggestion comes from a conversation with Breck Baldwin at StanCon 2019).

Many research users of PPLs have been mathematicians, meaning that the "natural" conceptualisations they are working with may be relatively sophisticated in mathematical terms. But if future applications of PPLs are in end-user tasks, should educational priorities shift from support for people who are familiar with the abstract operations, to those who have to treat the models and inferences as black box behaviour? What is the minimum conceptual framework for thinking about the behaviour of Bayesian models, and might a minimum framework be appropriate in school education?

Hitron et al (2019) investigated a "black box" approach, providing students with an experimental environment in which they were able to collect and label (gesture) data, train a classifier, and evaluate the resulting system behaviour. Through experimentation with the system, students did gain improved understanding of supervised learning. We should consider such results in relation to the ICT/computer science debate in school curriculum. The "ICT" perspective was that it was sufficient for students to know how to use applications like Powerpoint or Word, and not necessary to understand how these work internally (i.e. to learn programming). This policy has now been overturned, in favour of teaching CS at a more fundamental level, employing concepts that were previously not encountered until university level. On which side of this dichotomy might machine learning fall in future? Will training and using an ML system (for example, predictive text, or spam filtering) be a routine everyday task analogous to the use of Word or Powerpoint, or will it be a sophisticated intellectual task, providing a conceptual foundation for science and engineering? Will students benefit from being able to build new classifiers (using a PPL), or should a standard model be used to describe behaviour, for example in terms of feature selection, convergence, stability and generalisation?

Much of this prior research has focused on teaching PPL principles to people who might be expected to have some familiarity with principles of Bayesian probability. However, even communicating these principles may be challenging. Educational authorities Gage and Spiegelhalter ask 'Why, then, do people find [probability] so unintuitive and difficult? Well, after years of working in this area, we have finally concluded that this is because … probability *is* unintuitive and difficult.' (2018, p. 2). Can interactive construction and visualisation of PPL models help novices to gain an improved understanding of basic principles of conditional probability?

Here we briefly present an approach to visualizing probabilistic computation that is based on using spatial partitions as a visual metaphor for illustrating *discrete* probabilistic values. In this representation, we divide a rectangle into blocks, one for each value of the probabilistic value, so that the area occupied by each value corresponds to its probability (Erwig and Walkingshaw 2013). This notation captures three important features of a probabilistic value, namely (A) the fact that it may consist of multiple values, (B)

that each value has a distinctive probability associated with it, and (C) that the probabilities of all values sum up to 1. For example, the result of a fair coin flip is represented in this notation as in Fig. 2.

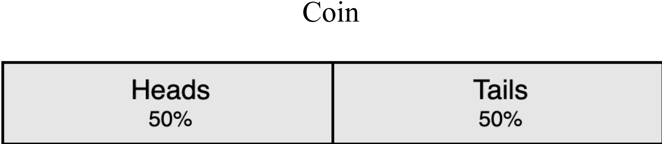

Figure 2: Visualisation of the space of possible values for a fair coin toss, partitioned into two blocks of equal size

Note that showing the value probabilities, while helpful to users, is redundant as far as the spatial representation goes, since they are derived from the relative sizes of the partition blocks.

This notation directly supports several operations on probabilistic values. Finding the probability of an event amounts to measuring the size of the occupied area. The computation of a joint probability involves superimposing rectangles so that the combination of values "shares" the common space. An example of how this visualisation is constructed for two coin flips is shown in Fig. 3 below.

The creation of the resulting partition can be broken down into a sequence of three basic spatial operations (Fig. 3). (1) Create a copy of the second partition for each value in the first partition. (2) Shrink each such copy horizontally to the width of the block of its corresponding value from the first partition. (3) Intersect the two partitions, and annotate the blocks of the resulting partition with the combination of the values (and the product of their probabilities). The computation of marginal probability distributions can be illustrated by a flow diagram that links the original to the result partition. Arrows indicate a spatial union operation of all blocks that have the same value once a value has been marginalized out.

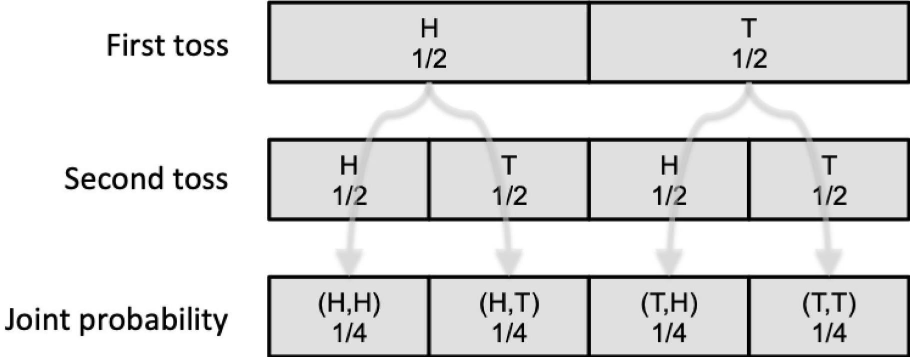

Figure 3: Construction of visualisation of a joint probability. The first coin toss has two equal partitions as in Fig. 2. The second toss is represented by two copies of that equally partitioned space, which have been shrunken horizontally to the same width as each original partition. The combination of the first and second values in the third row can be thought of as superimposing the first and second rows, with annotations now showing the combined values and product of probabilities.

We illustrate how this notation can assist in probabilistic reasoning by a simple example. Consider the following scenario: After throwing two coins, we are told that one of them came up Tails. What is the probability that the other one is Heads? Many people will state that the probability is 50%, whereas it is actually 67%. We can illustrate this computation by starting with a joint probability for two coins (Fig 4.). The first step is to select the blocks that correspond to the event "one came up Tails," which leaves three blocks, because we exclude the one containing two Heads. These three blocks define the probability space against which the query "What is the probability that the other one was Heads?" is posed. In the second step, the exclusion of the (H,H) block requires a resizing of the remaining blocks to occupy the whole

probability space, which leads to a partition with 3 blocks that each have an associated probability of ⅓. The third step requires the grouping of all blocks that match the query "What is the probability that the other one was Heads?" It is easy to see from Fig. 4 that this event corresponds to two blocks, which together occupy ⅔ of the partition.

Many probabilistic programs are using these basic operations as building blocks, and we can, in principle, apply the partition-based explanation mechanism to explain the execution of such programs. In (Erwig and Walkingshaw 2013) we have used this approach to explain linear programs employing a story-telling metaphor (Erwig 2017), but this approach could also be used to explain non-linear representations as used in probabilistic reasoning in Bayesian networks.

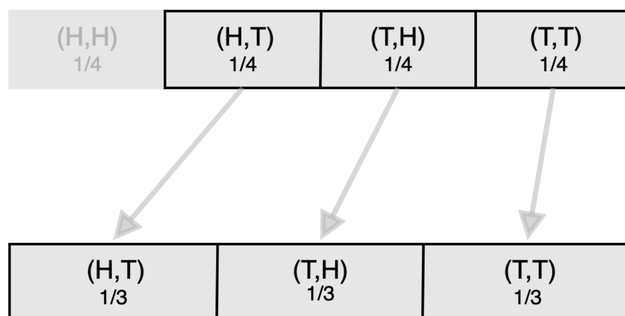

Figure 4: Reasoning about conditional probability distributions to calculate the probability that, given one of two coins has come up Tails, the other one was Heads. The observation that one of the coins came up tails eliminates the case (H,H). The remaining three cases are expanded to show that each has a probability of ⅓, with two of these three cases including Heads.

Usability challenges arising due to approximations in modeling and inference

In addition to the usual challenges arising from bugs in code, probabilistic programmers need to grapple with new and fundamental gaps between their intentions and the behaviors produced by executing their program. First, there is the widely known aphorism that "all models are wrong, but some models are useful" --- that is, the data generating process in the world that produces input data will not, in general, exactly correspond to the modeling assumptions encoded in the probabilistic program. At best, a generative model will be a useful approximation to the true data generating process. Second, the inference algorithms used in probabilistic programming systems are often approximate, and the quality of those approximations (and the tradeoffs between runtime, memory cost, and approximation accuracy) are often murky, both in theory and in practice. Thus, at best, inference results will be a useful approximation to true Bayesian inference in a model that is already itself at best an approximation to the true data generating process.

How can we help probabilistic programmers inspect, test, debug, and improve their probabilistic programs, in light of these difficulties? Cusumano-Towner and Mansinghka (2017) introduced a class of algorithms for estimating the accuracy of inference algorithms. Also, Saad, Freer, and Mansinghka (2019) introduced new multivariate goodness-of-fit testing techniques that can help probabilistic programmers use synthetic data to check the convergence of a broad class of Monte Carlo algorithms. These techniques can potentially be combined, to help probabilistic programmers build up a picture of the accuracy of inference on synthetic data. What about model mis-specification? Recently, techniques for posterior predictive checking, where simulated data from inference is compared to real data, to help detect model mis-specification, have been incorporated into probabilistic programming platforms such as PyMC3. However, these techniques only begin to address the fundamental inferential issues in checking models and estimating inference algorithm accuracy.

What user interfaces could help probabilistic programmers correctly apply these techniques? What distributed computing support is needed, to make it easy for a typical probabilistic programmer to run thousands of synthetic data experiments, as these techniques require? Could probabilistic modeling help probabilistic programmers extrapolate from synthetic tests, to predict inference accuracy on real data? What techniques could help probabilistic programmers detect model mis-specification, statically, before running an interactive application? Is it possible to write probabilistic programs that can detect model mis-specification dynamically, as new data is encountered? Probabilistic programmers urgently need answers to these questions.

A furthest-first initiative: AI tools for African students and researchers

A common strategy in other fields of usability research and human-centric system design is to consider the "furthest first". This identifies the class of users who are least well served by the current generation of technology or user interface, and gives priority to meeting the needs of those people. It often turns out that a design strategy focused on those who are least well-served results in benefits for all users. Perhaps the most dramatic example of this strategy in programming language research was the Smalltalk language, initially proposed as part of the KiddiKomp project at Xerox PARC (Kay 1996), as one of the first programming languages that would be accessible to children. As it turned out, Smalltalk was more popular among adult programmers than among children (although the underlying principles did continue to benefit very young programmers, in particular through the Smalltalk architecture that underpins the Scratch language). But an even more dramatic outcome of the Smalltalk project was the way in which it required the developers to rethink many other aspects of the programming user interface, leading to the invention of icons, windows, menus, and many other elements of the modern GUI. A furthest-first approach to programming language research can have extraordinarily far-reaching impact.

One of the authors (Blackwell) is currently planning a year-long project, investigating the requirements for probabilistic programming among a population that are currently not well-served by existing languages for AI and data science research. He plans to collaborate with programmers, end-users and students in four different African countries (Uganda, Kenya, Ethiopia and Namibia). This builds on work by author Church and others (e.g. Church, Simpson et al 2018) designing new tools and architectures for social science, public health and humanitarian research using text data obtained via SMS from regions with poor communications infrastructure. The application of AI methods in such contexts is often intended to empower local actors rather than follow the typical business models of software start-ups, meaning that greater access to configuration and control through accessible programming languages could be particularly important. Economic and political models may also differ from those in typical software technology contexts, for example considering whether those who contribute cognitive labour as a condition of access to media should be paid for their work (Blackwell 2019). The specific aspirations of people in low income countries are also likely to be different, in shaping the imagination of what AI systems can possibly do - providing tools that allow people to explore their own imaginative ideas therefore offers support for innovations that might not have been anticipated in corporate laboratories or universities in wealthy countries.

Some research issues that may be productive in these furthest-first contexts include:

- Redraw system boundaries to consider interaction between labelling and modelling
- Consider reform of postcolonial school curricula in maths and probability
- Explore AI as enabling structural innovation, not data science as statistical bureaucracy
- Acknowledge the economic and political tensions in cognitive labour such as labelling

We have already noted the specifically educational challenges and opportunities in the design of PPLs. These may present differently in low-income countries, and in relation to the mathematics curriculum taught in those countries. One interesting possibility is the role of probabilistic models in public discourse and activism, for example Carroll and Rosson's investigation of end-user development practices as part of participatory design for community informatics (2007). Experiments such as use of AgentSheets to discuss local community policy (Arias et al 1999) demonstrate the ways that simulation models might be integrated into other social contexts. We might describe this as "broad learning" in contrast to "deep learning", where a wider range of people are able to participate in the definition of models, rather than simply providing training data labels.

If teaching resources are limited, we might also consider following the design strategy of Sonic Pi and other educational languages, in which all tutorial content is integrated into the IDE itself. Sonic Pi has been successfully applied in an African context during the CodeBus Africa project that toured schools in 10 African countries over 100 days in 2017 (Bakić et al 2018).

Conclusion

This discussion represents work in progress. Although some initial advances have been reported here (summarising earlier publications), this paper should primarily regarded as a manifesto for promising research directions in the development of more usable PPLs. Several of the authors have substantial research projects in progress, and readers interested in this topic are encouraged to follow developments from those who have contributed.

A key message emerging from our discussions is that the core principles in PPLs are going to be relevant to several very different classes of programmer, and that each of these classes will have very different usability requirements. At present, most users of PPLs are researchers. Researchers do have usability needs, including straightforward considerations of effective software engineering tools (debuggers, tracers, smart editors and so on). It would be possible to carry out more comprehensive task analysis of research work processes, for example as in Marasoiu's study of data scientists (2017), to identify the activity profiles of these researchers and identify ways to optimise tools and notations that suit those profiles. It would also be possible to study the design representations that they already use, and integrate versions of these more closely into data science tools (for example, as in author Gordon et al's (2014) Tabular alternative to "plates and gates" diagrams). A second class of programmer is the person who needs to define, explore and apply probabilistic models, but is unlikely to have specialised training (the end-user case). For this class, building on familiar representation conventions such as spreadsheets or databases is likely to be valuable, in addition to supporting more casual and data-centric workflows. A third class is the pedagogic application, where the users are students acquiring an understanding of data science methods or even simple principles of Bayesian probability. For this class, conceptual clarity, minimal distracting syntax, and correspondence to naturalistic descriptions is likely to be important. Because most classroom programming exercises are built once and discarded, language considerations that are essential for software engineering (such as project documentation, design traceability, configuration management and performance optimisation) may be distracting rather than helpful. A repeated lesson from the past is that uncritical expectation for certain features to be "intuitive" for all users and application can be damaging for both educational and professional users, so we hope to avoid such naivety.

A Probabilistic Postscript

Two authors of this paper (Advait Sarkar and Tobias Kohn) were not able to attend the PPIG workshop where it was presented, because they got married on the same day. Not to each other. We invite readers to create a PPL model that would estimate the prior likelihood of such an event, for any given research publication, and thus assess the risk that this might continue to be a risk at future PPIG workshops. We also record our congratulations to Advait and Tobias!

References

Arias, E. G., Eden, H., Fischer, G., Gorman, A., & Scharff, E. (1999, December). Beyond access: Informed participation and empowerment. In Proceedings of the 1999 conference on Computer support for collaborative learning (p. 2). International Society of the Learning Sciences.

Bakić, I., Puukko, O., Hämäläinen, V., and Subra, R. (2018). CodeBus Africa Study. Aalto Global Impact; Aalto University. Washington, DC : World Bank Group. http://documents.worldbank.org/curated/en/357931528940784006/Codebus-Africa-Study

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

Baydin, A. G., Shao, L., Bhimji, W., Heinrich, L., Meadows, L. F., Liu, J., Munk, A., Naderiparizi, S., Gram-Hansen, B., Louppe, G., Ma, M., Zhao, X. Torr, P. H. S., Cranmer, K., Lee, V., Prabhat, Wood, F. (2019). Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19), November 17–22, 2019.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradham, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N. D. (2018): Pyro: Deep universal probabilistic programming. Journal of Machine Learning Research, 2018.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.

Blackwell, A.F. (2006). Psychological issues in end-user programming. In H. Lieberman, F. Paterno and V. Wulf (Eds.), End User Development. Dordrecht: Springer, pp. 9-30

Blackwell, A.F. (2017). End-user developers - what are they like? In F. Paternò and V. Wulf (Eds). New Perspectives in End-User Development. Springer, pp. 121-135.

Blackwell, A.F. (2019). Artificial intelligence and the abstraction of cognitive labour. In M. Davis (Ed.), Marx200: The significance of Marxism in the 21st century. London: Praxis Press, pp. 59-68.

Blei, D. M., Kucukelbir, A., McAuliffe, J. D.. (2017). Variational inference: A review for statisticians. Journal of the American Statistical Association, 112(518):859–877.

Borghouts, J., Gordon, A., Sarkar, A., Toronto, N. (2019). End-user probabilistic programming. Proceedings of the 16th International Conference on Quantitative Evaluation of Systems. Springer LNCS 11785:3-24.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., & Riddell, A. (2017). Stan: A probabilistic programming language. *Journal of statistical software*, *76*(1).

Carroll, J. M., & Rosson, M. B. (2007). Participatory design in community informatics. Design studies, 28(3), 243-261.

Carroll, J.M., Rosson, M.B., Isenhour, P., Ganoe, C., Dunlap, D., Fogarty, J., Schafer, W. and Van Metre, C., 2001. Designing our town: MOOsburg. International Journal of Human-Computer Studies, 54(5), pp.725-751.

Cheng, P. C. H. (2011). Probably good diagrams for learning: representational epistemic recodification of probability theory. Topics in Cognitive Science, 3(3), 475-498.

Church, L., Simpson, A., Zagoni, R., Srinivasan, S. and Blackwell, A.F. (2018). Building socio-technical systems for representing citizens voices in humanitarian interventions. In S. Tanimoto, S. Fan, A. Ko and D. Locksa (Eds), Proceedings of the Workshop on Designing Technologies to Support Human Problem Solving. University of Washington. pp. 19-21.

Cusumano-Towner, M., Saad, F., Lew, A., and Mansinghka, V., (2019). Gen: a general-purpose probabilistic programming platform with programmable inference. PLDI 2019.

Cusumano-Towner, M., and Mansinghka, V. (2017). AIDE: an algorithm for measuring the accuracy of probabilistic inference algorithms. NeurIPS 2017.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., Saurous, R. A. (2017): Tensorflow distributions. arXiv preprint arXiv:1711.10604.

Du Boulay, B. (1986): Some Difficulties of Learning to Program. Journal of Educational Computing Research, 2(1), pp. 57-73.

Edge, D., & Blackwell, A. (2006). Correlates of the cognitive dimensions for tangible user interface. Journal of Visual Languages & Computing, 17(4), 366-394.

Erwig, M. (2017). *Once Upon an Algorithm - How Stories Explain Computing*. MIT Press, Cambridge, MA.

Erwig, M. and Walkingshaw, E. (2013). A Visual Language for Explaining Probabilistic Reasoning. *Journal of Visual Languages and Computing, Vol. 24*, No. 2, 88-109.

Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., & Gelman, A. (2019). Visualization in Bayesian workflow. *Journal of the Royal Statistical Society: Series A (Statistics in Society)*, 182(2), 389-402.

Gage, J. and Spiegelhalter, D. (2018). *Teaching Probability.* Cambridge University Press.

Gelman, A., Lee. D., Guo, J. (2015): Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioral Statistics*, 40(5):530-543, 2015.

N. D. Goodman, J. B. Tenenbaum, and The ProbMods Contributors (2016). *Probabilistic Models of Cognition (2nd ed.)*. Retrieved 2019-6-25 from https://probmods.org/

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., Tenenbaum, J. B.(2012): Church: a language for generative models. arXiv preprint arXiv:1206.3255.

Gorinova, M.I., Sarkar, A., Blackwell, A.F. and Syme, D. (2016). A Live, Multiple-Representation Probabilistic Programming Environment for Novices. In Proceedings of CHI 2016, pp. 2533-2537.

Tom Hitron, Yoav Orlev, Iddo Wald, Ariel Shamir, Hadas Erel, and Oren Zuckerman. 2019. Can Children Understand Machine Learning Concepts?: The Effect of Uncovering Black Boxes. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19). ACM, New York, NY, USA, Paper 415, 11 pages. DOI: https://doi.org/10.1145/3290605.3300645

Hoffman, M. D., Gelman, A. (2014). The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. Journal of Machine Learning Research, 15(1):1593–1623.

Gabry, J., Simpson, D., Vehtari, A., Betancourt, M., & Gelman, A. (2019). Visualization in Bayesian workflow. Journal of the Royal Statistical Society: Series A (Statistics in Society), 182(2), 389-402.

Gordon, A. D., Graepel, T., Rolland, N., Russo, C., Borgstrom, J., & Guiver, J. (2014). Tabular: a schema-driven probabilistic programming language. In ACM SIGPLAN Notices (Vol. 49, No. 1, pp. 321-334). ACM.

Alan C. Kay. 1996. The early history of Smalltalk. In History of programming languages---II, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (Eds.). ACM, New York, NY, USA 511-598. DOI: https://doi.org/10.1145/234286.1057828

Kim, J., and Pearl, J. (1983). A computational model for causal and diagnostic reasoning in inference systems. In Proc. International Joint Conference on Artificial Intelligence (IJCAI), pp. 190-193.

Ko, A.J., Abraham, R., Beckwith, L., Blackwell, A.F., Burnett, M., Erwig, M., Lawrence, J., Lieberman, H., Myers, B., Rosson, M.-B., Rothermel, G., Scaffidi, C., Shaw, M., and Wiedenbeck, S. (2011). The State of the Art in End-User Software Engineering. ACM Computing Surveys 43(3), Article 21.

Kohn, T., Komm, D.: Teaching Programming and Algorithmic Complexity with Tangible Machines. In: Pozdniakiv, S., Dagien, V. (eds): Informatics in Schools. Fundamentals of Computer Science and Software Engineering. ISSEP 2018. Lecture Notes in Computer Science, vol. 11169, Springer, Cham.

Kulkarni, T., Kohli, P., Tenenbaum, J., and Mansinghka, V. Picture: a probabilistic programming language for scene perception. CVPR 2015.

Le, T. A., Baydin, A. G., Wood, F. (2017): Inference compilation and universal probabilistic programming. In Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS), volume 54 of Proceedings of Machine Learning Research, pages 1338–1348, Fort Lauderdale, FL, USA, 2017. PMLR.

Lieberman, H., Paternò, F. and Wulf, V. (2006). *End-User Development.* Springer.

Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. 13th Australasian Computer Education Conference (ACE 2011).

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

Mansinghka, V., Schaechtle, U., Handa, S., Radul, A., Chen, Y., and Rinard, M. Probabilistic programming with programmable inference. In PLDI 2018.

Mansinghka, V., Tibbetts, R., Baxter, J, Shafto, P., Eaves, B. BayesDB: A probabilistic programming system for querying the probable implications of data. arXiv:1512.05006 (2015).

Mărăşoiu, M. and Blackwell, A.F. (2017). User experiences in a visual analytics business In Proceedings of PPIG 2017 - the 28th Annual Workshop of the Psychology of Programming Interest Group.

Minka, T., Winn, J., Guiver, J., Knowles, D. (2013): Infer.net 2.4, Microsoft Research Cambridge. URL: http://research. microsoft. com/infernet.

Pane, J. F., Myers, B. A., & Garlan, D. (2002). A programming system for children that is designed for usability (Doctoral dissertation, School of Computer Science, Carnegie Mellon University). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.481.2364

Pearl, J., & Mackenzie, D. (2018). The book of why: the new science of cause and effect. Basic Books.

Pennington, N. (1987). Comprehension strategies in programming. In Empirical studies of programmers: second workshop (pp. 100-113). Ablex Publishing Corp..

Pennington, N., Lee, A. Y., & Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. Human-Computer Interaction, 10(2), 171-226.

Peyton Jones, S., Blackwell, A and Burnett, M. (2003). A user-centred approach to functions in Excel. In Proceedings International Conference on Functional Programming, pp. 165-176.

Repenning, A., & Ambach, J. (1996). Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In Proceedings 1996 IEEE Symposium on Visual Languages (pp. 102-109). IEEE.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J.S., Silverman, B. and Kafai, Y.B., (2009). Scratch: Programming for all. Comm. ACM, 52(11), 60-67.

Saad, F. and Mansinghka, V. (2016). A probabilistic programming framework for probabilistic data analysis. NeurIPS 2016.

Saad, F. and Mansinghka, V. (2017). Detecting dependencies in multivariate databases with probabilistic programming and non-parametric Bayes. AISTATS 2017.

Saad, F., Freer, C., Ackerman, N., and Mansinghka, V. (2019). A family of exact goodness-of-fit tests for high-dimensional discrete distributions. AISTATS 2019.

Scaffidi, C., Shaw, M., & Myers, B. (2005). Estimating the numbers of end users and end user programmers. In 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05) (pp. 207-214). IEEE.

Spiegelhalter, D., Thomas, A., Best, N., Gilks, W. (1996): BUGS 0.5: Bayesian Inference Using Gibbs Sampling Manual (version ii). MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK, pages 1–59,

Stan Development Team. (2017). ShinyStan: Interactive visual and numerical diagnostics and posterior analysis for Bayesian models. *R Package Version*, *2*.

Staton, S., Wood, F., Yang, H., Heunen, C., Jammar, O. (2016): Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In 2016 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), pages 1–10. IEEE, 2016.

Stead, A., & Blackwell, A. F. (2014). Learning syntax as notational expertise when using drawbridge. In Proceedings of the Psychology of Programming Interest Group Annual Conference (PPIG 2014) (pp. 41-52).

Streit, A. (2008). Encapsulation and abstraction for modeling and visualizing information uncertainty. PhD thesis, Queensland University of Technology.

Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., Blei, D. M. (2017): Deep probabilistic programming. arXiv preprint arXiv:1701.03757

Van de Meent, J.-W., Paige, B., Yang, H., Wood, F. (2018): An Introduction to Probabilistic Programming. arXiv e-prints, Sep 2018

Watt, S. (1998). Syntonicity and the psychology of programming. Proceedings of PPIG 1998.

Wingate, D., Stuhlmueller, A., Goodman, N. (2011). Lightweight implementations of probabilistic programming languages via transformational compilation. In Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, pages 770–778.

Wood, F., Van de Meent, J.-W., Mansinghka, V. (2014): A new approach to probabilistic programming inference. In Artificial Intelligence and Statistics, pages 1024–1032, 2014.

Zhou, Y., Gram-Hansen, B. J., Kohn, T., Rainforth, T., Yang, H., Wood, F. (2019): LF-PPPL: A low-level first order probabilistic programming language for non-differentiable models. arXiv preprint arXiv:1903.02482

Appendix - technical discussion of the declarative/imperative/generative distinction

During collaborative drafting of the introductory section of this paper, we found considerable difference of opinion among the authors, with regard to a rather basic point - how to define the computational class of languages that PPLs fall into. A simple definition by reference to earlier research at PPIG could have referred to advocacy for declarative languages in the 1980s and 1990s, but there were many important technical distinctions - far too many to include in the introduction to a paper on usability! The following appendix records the comment thread in order to capture the variety of issues that were raised in this technical discussion, many of which will be of interest to readers wishing to carry out technical experiments in the field.

Maria Gorinova

Some languages (e.g. Pyro and Edward) are of generative nature --- the idea is that the program corresponds to some generative process that describes the way the data was generated. E.g. I roll a standard 6-sided dice and write it in a variable n. I then roll an n-sided dice and write the result in a variable y. y is 4, what are my believes about x? In my opinion, this is rather imperative, and seems to be treated as such in science applications (e.g. describe the evolution of the universe based on some unknown cosmological parameters and a particular hypothesis, and then observe the 2D picture of stars as seen from Earth to figure out if the hypothesis is consistent with the data; https://arxiv.org/abs/1701.00478)

Depending on what one's mental model regarding a Stan program is, Stan can be seen as either declarative or imperative too. A Stan program can be thought of as encoding different factors in the joint probability distribution over all random variables. In this case it can be said it's declarative. But a Stan program can also be understood as an imperative (and deterministic) function that computes the joint probability density at a particular point. Many Stan developers and users think about their programs in this way, and thus would argue that Stan is fundamentally imperative.

Tobias Kohn

I really like this distinction between generative and declarative. I wonder, though, if the generative way is something like a "meta-declarative" way, where the model is generated through execution of the code.

I am thinking here in terms similar to how you model an object in OOP. In a language like Java, this is done declarative and rather statically, where you clearly specify each field to be in the class/object at "compile time" if you will. In dynamic languages like Python or JavaScript, however, an object (or even a class) is created through the execution of program code, allowing for dynamic generation of fields.

Nonetheless, the outcome is still about the same in that we have some sort of model afterwards.

As far as I understand, probabilistic systems in Python naturally tend to be in a generative way, where the model is created through execution of the program. Packages like TensorFlow then create internally a full blown computation graph or model, which can then be inspected and manipulated.

In all of this, I feel that the imperative features are more accidental because of the chosen language or method for creating the model. But even for very large and complex models such as the cosmological example, there is an actual model there that is at least implicitly expressed through the language.

Nonetheless, I feel that working out distinctions such as the generative vs. declarative way of writing models can be extremely helpful for novice programmers in that field, and we should pick it up as part of our discussion concerning design and education.

The description of the two ways of how to look at a Stan program is very nice. As I understand, the second way of looking at it is as a (computable) function that can be run to obtain values. However, the "imperative" nature of this is not really necessary, right? We could just have the very same idea in a purely functional language without imperative features?

Maria Gorinova

Wow, very interesting discussion, indeed! Thank you so much for the comments, this is very thought provoking. I see what you mean --- the code is always there to be transformed into an inference algorithm, rather than describing the inference algorithm (the execution) itself. I would still argue that PPLs can also be conceptually imperative or functional, and not necessarily in a meta-declarative way.

Take for example the Lotka-Volterra population model. It concerns the evolution of the population of predators and the population of prey. Suppose that we start with X predators and Y prey, and after time T we have X_T predators and Y_T prey. The parameters of the model p1, p2, p3 and p4 control the rate of predators births, predators deaths, prey births and prey deaths respectively. We are interested in performing inference on these parameters given X, Y, X_T and Y_T. A probabilistic program describing the evolution of the system can then be written (in rough pseudo-code) as follows:

```
// observe initial size of the populations

observe(x = X, y = Y)

t = 0 // start at time 0

while(t < T){

// sample time of next interaction

dt ~ p(t)


// sample i --- the interaction that actually happens

i ~ categorical([exp(p1)*x*y, exp(p2)*x, exp(p3)*y, exp(p4)*x*y])

if (i = 1) x += 1 // predator birth

if (i = 2) x -= 1 // predator death

if (i = 3) y += 1 // prey birth
```

if (i = 4) y -= 1 // prey death

// go forwards in time

t += dt

}

// observe actual population size at time T

observe(x = X_T, y = Y_T)

```

## Maria Gorinova

There is a probabilistic model behind it indeed, but this model literally is the generative process given by the program. There is some random number of random variables inside the program. We can't create a factor graph or other graphical model describing this. We can't even write down the likelihood of this model in a meaningful way (the likelihood is intractable here). Inference methods that are applicable to this model would literally run the simulation forwards many times, either rejecting impossible runs (e.g. approximate Bayesian computation --- quite inefficient), or using neural networks to approximate the posterior or likelihood (e.g. sequential neural likelihood --- https://arxiv.org/abs/1805.07226). So the model is the generative process given by the program.

To the best of my knowledge, such programs can be written in Python-based PPLs, and also the functional Anglican. Edward and Pyro indeed get transformed into a Tensorflow/PyTorch computation graph, but in the newest version of both languages, this graph is dynamic.

Regarding Stan: yes, one of the ways to see a Stan program is as specifying a computable function that evaluates the (log) joint density of the model for particular values of the parameters. For example, if we have a model with fixed data x and parameters z, the Stan code really is a C-like function of z, which returns a real number --- the density at that z: real f(z) {return log p(z, x);}. That function truly is imperative. Parts of it are executed as is during inference (the Stan compiler is written in C++). But one could argue that the actual Stan program isn't, as it definitely possesses some declarative constructs (e.g. the different program blocks). I'm not sure what to think about it either!

Is the imperative nature of this necessary and can we write this function in a functional language: yes, but we can write any computation by using, say, the lambda-calculus. It just might be very messy for some programs, I suppose. Here is a good real-life example of a Stan program, for which is not immediately obvious to me how to re-formulate in a functional manner, due to some random access to arrays, etc:https://github.com/pkremp/polls/blob/master/state%20and%20national%20polls.stan

## Tobias Kohn

Thank you very much for your reply and this great discussion!

Let me perhaps briefly elaborate what I was trying to say concerning the "imperative" nature of programs.

If you look at a silly little function like

def foo(x):

x += 1

result = [ ]

result.append(x)
```

```
        return result
```

Then the function's body is clearly written in an imperative style. But I would call this "imperative"ness rather accidental as it is of absolutely no importance to the "outside" world using the function.

My thought was that a prob prog program cannot really be fundamentally imperative because we want to run and rerun it for simulation and inference purposes. When the program is intrinsicaly imperative in that it changes some external state of the machine, each rerun takes place under different conditions. Hence, I concluded that this cannot really be what you meant with "imperative".

If I understand you correctly, then you argue that some programs are intrinsically imperative because the model evolves with time based on the current state it is in (such as the Lotka-Volterra), making random access to memory and data. This is an excellent point. I wonder, however, if in the end we might actually talk about the same thing and just consider different levels of abstraction. Keeping the example of the function above in mind: do you think that prob programs can always be encapsulated in a "pure" function, or are there cases where running the program alters a state that is important for the next rerun? In other words: are there any prob programs for which it makes sense to discover how often they have already run?

(this is a serious question and not a rhetorical one)

Your Lotka-Volterra example is really nice, and I like it (as an example for our discussion). It makes an excellent point concerning the declarative vs. generative style.

Actually, I thought of "model" in a much more abstract sense than "graphical model" or "factor graph", etc. And I feel I can quite clearly see the "model" in the Lotka-Volterra, without being able to fully grasp it. Still, your example is intriguing and I shows that my view of generative programs as meta-declarative is rather incomplete and falls short.

In a way, each time we run the prob program, we get out a different instance of the underlying model, right? As you say, the real model itself is intractable here, defying any easy formulation as "graph" as we know it.

If you indulge me in trying to find some parallels to ground more familiar to me (I am a "compilers-guy", by the way ^_^)... is this not more or less the same thing a "constructor" in Python or JavaScript does? Each time I call it, I get back an instance of a class. However, because of the dynamic nature of these languages, these instances can differ greatly, and do not even have to have the same structure.

Mind, I am absolutely not sure if this is actually the right way of thinking about it at all. I am still convinced, though, that your initial distinction between declarative and generative is very important, and am trying to figure out how to properly process this...

Atılım Güneş Baydin

I just want to make the comment that I agree with Maria's comment at the top of this comment chain that probabilistic programs are of "generative nature --- the idea is that the program corresponds to some generative process that describes the way the data was generated. " This is exactly how I see them. A probabilistic program is an explicitly specified generative process, defining (1) a probabilistic generative model and (2) the associated joint prior distribution at the same time.

Maria Gorinova

Thank you so much for clarifying, Tobias! I think I understand much better what you mean now. I agree: it seems like we may be talking about the same thing, but considering different levels of abstraction. Indeed, I can’t think of a meaningful example where the model is a function with side effects. I also have no idea if something like that would be possible (though I can’t immediately think why it wouldn’t be either).

You are right about the similarity with constructors in Python, I think. An important difference though, is that in the case of constructors, they are called within the same language, as part of a bigger program. In probabilistic programming, that is only sometimes the case (all embedded PPLs, I suppose), while in other cases the program is standalone (e.g. Stan). I don’t know if that’s important at all.

Thank you again for an excellent discussion!

As to the generative vs declarative distinction: I wonder if there is some previous work that gives a good explanation of this. I can't think of anything off the top of my head. But it relates to the type of supported models. For example, if we say that declarative PPLs really specify the factorisation of the joint distribution of all variables, this corresponds to (I think) graphical models (including directed, undirected and mixed graphical models, but not arbitrary stochastic processes). While generative PPLs allow for a generative process to be described (including directed graphical models, but not undirected / mixed graphical models). If anyone knows of a good read describing this, please let me know --- I will be very grateful!

Vikash K. Mansinghka

Thanks for this discussion! Briefly:

1. I think that generative languages support both the "generative" and "declarative" modes you're describing.

WebPPL (and Venture, and Gen, among others) allows for mixed undirected and directed models. For example, WebPPL has both "sampling" statements and "factor" statements that add terms to the log density of an execution (but do not make explicit stochastic choices). Languages without "factor" statements can also support "undirected" constraints via adding coin flips to the model with weights corresponding to the factor to be multiplied into the joint probability density. This has been used in Venture in various applications.

2. I agree with earlier comments by Tobias that we need to draw a distinction between languages that try to provide "automated inference" (taking a "declarative" view) and languages that provide programmable inference constructs that let users customize inference (e.g. Venture and Gen, but also arguably Pyro, Edward, and Turing, to various degrees, and also platforms like TensorFlow).

In our own AI research, mostly in computer vision and Bayesian data analysis, it has been essential to be able to customize inference. Automated inference doesn't work well enough --- generic Monte Carlo and generic deep learning architectures can both fail on simple problems that are easy for custom algorithms to solve. This is one reason why we've worked hard to make languages that support programmable inference.

I hope the probabilistic programming community learns from the experience of earlier, logic-based AI programming languages. I think we need to try to find simple, teachable abstractions for programmable inference, that help future users of probabilistic programming visualize how the inference process unfolds, and how it depends on the generative process in the underlying model. It's especially important for users to be able to predict how much runtime and memory individual inference operations will consume. If, instead, we offer our users black-box "declarative" systems, I think they may be too hard for non-experts to learn.