Calculating distributions

Chung-chieh Shan
Computer Science
Indiana University
Bloomington, Indiana, USA
ccshan@indiana.edu

Abstract

The ways we reason about probability distributions and explore their applications have been naturally shifting: away from thoughtful proving with definitions using first principles, and towards mechanical calculation with expressions using derived principles. This talk reviews three useful operations on distributions that we have started to express using equational derivations and even to automate as program transformations. These operations are (1) to recognize a density function as belonging to a known distribution family, (2) to eliminate an unused random variable by summation or integration, and (3) to disintegrate a joint measure into a marginal and a conditional measure. It is thus promising to support probabilistic reasoning by drawing techniques from both programming languages and computer algebra. Ongoing challenges include how to handle a wide variety of container data types and generating programs, and how human guidance should interact with machine assistance.

CCS Concepts • **Mathematics of computing** → **Probabilistic representations**; *Variable elimination*; *Density estimation*; *Distribution functions*; *Statistical software*; • **Theory of computation** → *Automated reasoning*; *Denotational semantics*; • **Computing methodologies** → *Symbolic and algebraic manipulation*;

Keywords Probabilistic programs, conditional measures, disintegration

ACM Reference Format:
Chung-chieh Shan. 2018. Calculating distributions. In *The 20th International Symposium on Principles and Practice of Declarative Programming (PPDP '18), September 3–5, 2018, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236950.3236973

PPDP '18, September 3–5, 2018, Frankfurt am Main, Germany

ACM ISBN 978-1-4503-6441-6/18/09.
https://doi.org/10.1145/3236950.3236973

1 Introduction

Improving our understanding of a domain and automating reasoning in the domain are two activities that ideally form a virtuous cycle [2]. For example, by exploring the notion of natural numbers from first principles, we might come up with a collection of arithmetic operations and equational laws that are useful over and over again, and eventually develop representations such as decimal place-value and algorithms such as grade-school addition. Once we get the hang of invoking an automated reasoning procedure that is faster and more predictable if less flexible, we can redirect our thoughts to notions expressed using arithmetic and eventually automate reasoning with those notions. Similarly, we can use Gaussian elimination instead of thinking about which variables to cancel with which, and we can use a context-free parsing algorithm instead of searching for a syntactic derivation in a grammar.

Another domain where our improved understanding is leading to automated reasoning is probability distributions. Distributions are popular for formulating and solving problems in a wide variety of fields including machine learning, system control, and cognitive science. Many of these computations are automated, but the programs typically consist of manually derived formulas that operate on numbers, so any intent that relates programs to distributions is lost. We envision programs that refer to distributions directly and operate on them compositionally, using the same vocabulary as human practitioners communicating with each other. We expect such an executable vocabulary to make program families [14] that compute with distributions easier to express and evolve over time. This vision is one view of *probabilistic programming*.

2 Representing distributions

We must represent distributions before we can automate reasoning about them. To start, we write the informal type $\mathbb{M}\,\alpha$ for distributions (measures) over α. For example, the normal distribution over reals has the type $\mathbb{M}\,\mathbb{R}$.

It is popular to represent distributions as densities. For example, it is popular to regard the normal distribution over reals as a function $\mathbb{R} \to \mathbb{R}_+$. But a density only represents a distribution in conjunction with a *base measure* (or *dominating measure*), and leaving out the latter is like leaving out the unit on a physical quantity: it works at the beginning but becomes confusing.

Instead, we represent distributions using operations of a monad [7, 16]. Here we notate monad unit by **return** and monad bind by Haskell's do-notation, and add primitives for elementary distributions. For example, consider the following problem (adapted from Eddy's [5]):

> An unknown random process yields a stateless coin that can be flipped repeatedly to produce heads (H) or tails (T). We assume that the probability p that the coin produces H each time is distributed uniformly between 0 and 1 by the process. We flip the coin 8 times and observe THTHHTHH. What is the probability that the next flip produces H versus T?

Using the primitive distributions

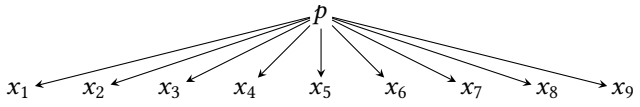
$$\textbf{uniform} : \mathbb{R} \to \mathbb{R} \to \mathbb{M}\,\mathbb{R} \tag{1}$$

$$\textbf{flip} : [0,1] \to \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}, \tag{2}$$

we represent the process producing 9 coin flips as follows:

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1;\\ &\quad x_1 \leftsquigarrow \textbf{flip}\ p; \ldots; x_9 \leftsquigarrow \textbf{flip}\ p;\\ &\quad \textbf{return}\ (x_1,\ldots,x_9)\} : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}^9\end{aligned} \tag{3}$$

Each of the 10 bindings ($\leftsquigarrow$) in this expression can be viewed as a node in a graph whose edges are data dependencies.



This directed acyclic graph is a *Bayes net* [15], so the monadic representation of distributions is essentially a Bayes net. One difference is that monadic bindings are ordered—for example, x_1 is chosen before x_9 in (3). But that is just a matter of bureaucracy: reordering bindings is justified by the semantics introduced by Staton [18], where measure terms denote well-defined *s-finite kernels*.

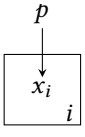
Two terms equivalent to (3) are worth mentioning on the side. First, for programmers comfortable with treating monadic actions as values, it may be more natural to represent the coin not by $p : [0,1]$ but by $c : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}$:

$$\begin{aligned}&\textbf{do}\ \{c \leftsquigarrow \textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1; \textbf{return}\ (\textbf{flip}\ p)\};\\ &\quad x_1 \leftsquigarrow c; \ldots; x_9 \leftsquigarrow c;\\ &\quad \textbf{return}\ (x_1,\ldots,x_9)\} : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}^9\end{aligned} \tag{4}$$

(Treating monadic actions as values helps represent their mixtures [6, 8, 11].) Second, the repeated calls to **flip** in (3) (or to c in (4)) can be coalesced using a loop combinator such as what Haskell calls **replicateM** : $\mathbb{Z} \to \mathbb{M}\,\alpha \to \mathbb{M}\,[\alpha]$:

$$\textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1; \textbf{replicateM}\ 9\ (\textbf{flip}\ p)\} \tag{5}$$

Repetition in a Bayes net can similarly be notated using *plates* [3] such as the rectangular box to the right. Handling repetition efficiently is called *lifted* inference and a research topic for both Bayes nets and probabilistic programs.



It remains to represent the observations THTHHTHH. Because the space of observations $\{\mathrm{H},\mathrm{T}\}$ is discrete (see Section 6), we can use the primitive *scoring* construct **factor** : $\mathbb{R}_+ \to \mathbb{M}\,\mathbb{1}$, which maps 0 to the zero measure and 1 to the trivial measure **return** ().

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1;\\ &\quad x_1 \leftsquigarrow \textbf{flip}\ p; \ldots; x_9 \leftsquigarrow \textbf{flip}\ p;\\ &\quad () \leftsquigarrow \textbf{factor}\ (\textbf{if}\ (x_1,\ldots,x_8) = (\mathrm{T},\mathrm{H},\mathrm{T},\mathrm{H},\mathrm{H},\mathrm{T},\mathrm{H},\mathrm{H})\\ &\qquad\qquad \textbf{then}\ 1\ \textbf{else}\ 0);\\ &\quad \textbf{return}\ x_9\} : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}\end{aligned} \tag{6}$$

The use of **factor** zeros out the portion of the distribution that does not match the observed 8 flips, and the **return** x_9 at the bottom makes this expression denote a distribution over just the outcome of the 9th flip.

3 Sampling and integrator semantics

One way to understand the meaning of a program using **factor** like (6) is that it can be directly executed as a sampler that collects samples of x_9 weighted by the scores. These weighted samples can then be analyzed; for example, to estimate the conditional probability of x_9 given $x_1,\ldots,x_8$, we can plot a histogram by totaling the weights (not counting the samples) in each of the two bins. In short, the program (6) can be interpreted as a *rejection sampler*. But to rewrite these programs to more efficient ones, we need to allow ourselves to preserve a coarser semantics than weighted sampling.

Running the program (6) as a weighted sampler is one way to estimate the conditional probability of x_9. But instead of collecting lots of samples weighted by 0 or 1, we can reduce the variance of the estimator—that is, make its accuracy depend less on chance—by rewriting it to generate weights other than 0 or 1. It is intuitive to replace drawing x_1 then testing $x_1 = \mathrm{T}$ by scoring $1 - p$. Hence, we get the same histogram on average when we replace (6) by

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1;\\ &\quad () \leftsquigarrow \textbf{factor}\ p^5(1-p)^3;\\ &\quad x_9 \leftsquigarrow \textbf{flip}\ p; \textbf{return}\ x_9\} : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}.\end{aligned} \tag{7}$$

This new program only draws p and x_9 randomly. It is an example of an *importance sampler*.

Rewriting (6) to (7) does not preserve weighted-sampling semantics but does preserve measure semantics. A measure over α is characterized by how it integrates functions $\alpha \to \mathbb{R}_+$ to yield results in $\mathbb{R}_+$. For example, **uniform** 0 1 integrates a function $f : \mathbb{R} \to \mathbb{R}_+$ to yield the result $\int_0^1 f(p)\,dp$. Similarly, (6) and (7) integrate a function $f : \{\mathrm{H},\mathrm{T}\} \to \mathbb{R}_+$ to yield the expected weight accumulated in a histogram bin. That expected weight is the same whether a sampler accumulates 1 in the bin 20% of the time or 0.2 in the bin 100% of the time, so (6) and (7) denote the same measure. Characterized thus as integrators, these measure denotations are easy to define compositionally on probabilistic programs.

4 Recognizing a density function

The first two lines of (7) express a distribution over p

$$\textbf{do}\ \{p \leftsquigarrow \textbf{uniform}\ 0\ 1;\ () \leftsquigarrow \textbf{factor}\ p^5(1-p)^3;\ \textbf{return}\ p\} \quad (8)$$

by naming the base measure **uniform** 0 1 and the density $p^5(1-p)^3$. As is common, the base measure represents our prior belief, and the density scores how likely the observed flips are under each possible value of p, so they are called the *prior* distribution and the *likelihood*, and together they define the *posterior* distribution (8).

It turns out that we can better understand the program (8) as a measure, and further reduce its variance as a weighted sampler, by rewriting it to the equivalent program

$$\textbf{do}\ \{() \leftsquigarrow \textbf{factor}\ (1/504);\ \textbf{beta}\ 6\ 4\}. \quad (9)$$

Here **beta** is a primitive distribution that, like **uniform** and **flip**, is well studied and comes with a sampling algorithm that doesn't even need to generate weights. The constant *normalizing* **factor** (1/504) is the total probability of our observed flips; it does not concern our belief about p.

Thanks to this *conjugacy* relationship between the prior and the likelihood on p, we can improve the program (7) by rewriting it to the equivalent program

$$\begin{aligned}&\textbf{do}\ \{() \leftsquigarrow \textbf{factor}\ (1/504);\\ &\quad p \leftsquigarrow \textbf{beta}\ 6\ 4;\\ &\quad x_9 \leftsquigarrow \textbf{flip}\ p;\ \textbf{return}\ x_9\} : \mathbb{M}\,\{\mathrm{H},\mathrm{T}\}.\end{aligned} \quad (10)$$

To automate this rewrite, a probabilistic programming system needs to recognize the formula $p^5(1-p)^3$ as a density of **beta** 6 4 with respect to **uniform** 0 1 up to a normalizing factor. Recognizing density formulas is a problem solved by *holonomic representation* in computer algebra [4, 9]. The basic idea is to characterize the function $h(p) = p^5(1-p)^3$ up to a factor by a *homogeneous linear differential equation*

$$g_n(p)\cdot h^{(n)}(p) + \cdots + g_1(p)\cdot h'(p) + g_0(p)\cdot h(p) = 0, \quad (11)$$

in which each $g_i(p)$ is a polynomial in p. In this case, the holonomic equation (unique up to a factor) is

$$p(1-p)\cdot h'(p) + \big(3p - 5(1-p)\big)\cdot h(p) = 0. \quad (12)$$

Because this equation can be computed compositionally, and polynomials and their ratios can be matched robustly, we can rewrite (7) to (10) without being stymied—like syntactic pattern matching would—by algebraic variations such as polynomial expansion and variable substitution. We have automated such rewriting for many one-dimensional primitive distributions, without hard-coding each conjugacy relationship. Thus, holonomy offers a promising representation of densities, closed under operations such as product and sum.

5 Eliminating a random variable

In the programs (6), (7), and (10), the last line **return** x_9 says that we want to predict the next flip but we don't care to learn about p (the coin itself). Consequently, a little calculus can help us improve the programs even more. Let (h, t) be $(1, 0)$ if x_9 is H and $(0, 1)$ if x_9 is T. Then the total weight of the outcome x_9 is

$$\int_0^1 p^5(1-p)^3 p^h (1-p)^t\, dp = \frac{\Gamma(6+h)\,\Gamma(4+t)}{\Gamma(10+h+t)}. \quad (13)$$

Specifically, the outcomes H and T have weights 6/5040 and 4/5040 respectively, so the program is equivalent to just

$$\textbf{do}\ \{() \leftsquigarrow \textbf{factor}\ (1/504);\ \textbf{flip}\ (6/10)\}. \quad (14)$$

The variable p has been eliminated, or *integrated out*. We have derived the exact solution to the original problem: the probability of the next flip is H 6/10 versus T 4/10.

In many real-world problems, such an exact solution is not tractable to compute. However, key to deriving an approximation algorithm is still exact equational reasoning, subject to human guidance in the foreseeable future [12, 19].

A computer algebra system can perform the integral (13) symbolically. But feeding a probabilistic program to computer algebra willy-nilly can easily make it worse, so automation requires judicious control over which integrals to try [4].

Another way to perform integrals that arise from probabilistic programs is to forego existing facilities for symbolic definite integration but to recognize the integrand in (13) by its holonomic representation to be a density of **beta** $(6+h)$ $(4+t)$ up to a normalizing factor. We can then use the fact that **beta** $(6+h)$ $(4+t)$ integrates to 1 (in other words, is a probability distribution), so the integral (13) is equal to the normalizing factor.

6 Disintegrating a joint measure

Consider the following problem, which is analogous to the running example so far but with real-valued observations:

> An unknown random process yields a stateless particle whose one-dimensional position can be measured repeatedly to produce a real number. We assume that the position p of the particle is distributed normally with mean 3 and standard deviation 2. We measure the particle 8 times, each time drawing independently from the normal distribution with mean p and standard deviation 1, and observe -1.4, $+1.0$, -0.2, -0.5, -1.4, $+0.9$, $+1.1$, -0.9. What is the distribution of the next measurement?

Using the primitive distribution

$$\textbf{normal} : \mathbb{R} \to \mathbb{R}_+ \to \mathbb{M}\,\mathbb{R}, \quad (15)$$

we can represent the process producing 9 measurements as in (3):

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{normal}\ 3\ 2;\\ &\quad x_1 \leftsquigarrow \textbf{normal}\ p\ 1;\ \ldots;\ x_9 \leftsquigarrow \textbf{normal}\ p\ 1;\\ &\quad \textbf{return}\ (x_1, \ldots, x_9)\} : \mathbb{M}\,\mathbb{R}^9\end{aligned} \quad (16)$$

However, it is no longer correct to represent the 8 observations using scores that are zero most of the time. The obstacle is that the probability of drawing any particular number from a normal distribution is exactly zero. Thus, when by analogy to (6) we write

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{normal}\ 3\ 2; \\ &\quad x_1 \leftsquigarrow \textbf{normal}\ p\ 1; \ldots; x_9 \leftsquigarrow \textbf{normal}\ p\ 1; \\ &\quad () \leftsquigarrow \textbf{factor}\ (\textbf{if}\ (x_1, \ldots, x_8) = (-1.4, +1.0, \ldots, -0.9) \\ &\qquad\qquad \textbf{then}\ 1\ \textbf{else}\ 0); \\ &\quad \textbf{return}\ x_9\} : \mathbb{M}\,\mathbb{R},\end{aligned} \tag{17}$$

we get a program equivalent to the zero measure. In other words, we might as well have written **factor** 0 instead.

To express the conditional distribution given $x_1, \ldots, x_8$, we need to use nonzero scores that are the densities of **normal** p 1 at x_i. Because **normal** is primitive, we can look up the density function (call it **dnorm** p 1) from a table:

$$\begin{aligned}&\textbf{do}\ \{p \leftsquigarrow \textbf{normal}\ 3\ 2; \\ &\quad () \leftsquigarrow \textbf{factor}\ \big(\textbf{dnorm}\ p\ 1\ (-1.4) \cdot \\ &\qquad\qquad \textbf{dnorm}\ p\ 1\ (+1.0) \cdot \cdots \cdot \\ &\qquad\qquad \textbf{dnorm}\ p\ 1\ (-0.9)\big); \\ &\quad x_9 \leftsquigarrow \textbf{normal}\ p\ 1; \textbf{return}\ x_9\} : \mathbb{M}\,\mathbb{R}\end{aligned} \tag{18}$$

This program is analogous to (7) directly and, thanks to a conjugacy relationship among normal distributions, amenable to the same equational reasoning as in Sections 4 and 5. These calculations are the basis of *Kalman filters* [10], which estimate the state of a system from measurements over time.

When an observed quantity is not drawn directly from a primitive distribution, but rather a result computed by the model (such as the center of mass of random particles [1]), it is more involved to specify and implement the *disintegration* program transformation turning (16) into (18) [13, 17]. In particular, we recently extended automatic disintegration to applications where the distribution of the observation has no density with respect to the Lebesgue base measure. For example, if the positions measured are *clamped* to the interval $[0, 1]$, so $x_i < 0$ is measured as 0 and $x_i > 1$ is measured as 1, we can still infer the distribution of p and x_9. In these cases, it is crucial for our representation of distributions to make explicit the base measure behind each density.

Acknowledgments

This research was supported by DARPA contract FA8750-14-2-0007, NSF grant CNS-0723054, Lilly Endowment, Inc. (through its support for the Indiana University Pervasive Technology Institute), and the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

References

[1] Hadi Mohasel Afshar, Scott Sanner, and Christfried Webers. 2016. Closed-Form Gibbs Sampling for Graphical Models with Algebraic Constraints. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.

[2] Bruno Buchberger. 2006. Mathematical Theory Exploration. In *Automated Reasoning: 3rd International Joint Conference (Lecture Notes in Computer Science)*, Ulrich Furbach and Natarajan Shankar (Eds.). Springer, 1–2. Invited talk.

[3] Wray L. Buntine. 1994. Operations for Learning with Graphical Models. *Journal of Artificial Intelligence Research* 2 (1994), 159–225.

[4] Jacques Carette and Chung-chieh Shan. 2016. Simplifying Probabilistic Programs Using Computer Algebra. In *Practical Aspects of Declarative Languages: 18th International Symposium, PADL 2016 (Lecture Notes in Computer Science)*, Marco Gavanelli and John H. Reppy (Eds.). Springer, 135–152.

[5] Sean R. Eddy. 2004. What is Bayesian Statistics? *Nature Biotechnology* 22, 9 (Sept. 2004), 1177–1178.

[6] Thomas S. Ferguson. 1973. A Bayesian Analysis of Some Nonparametric Problems. *The Annals of Statistics* 1, 2 (March 1973), 209–230.

[7] Michèle Giry. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis: Proceedings of an International Conference Held at Carleton University, Ottawa, August 11–15, 1981 (Lecture Notes in Mathematics)*, Bernhard Banaschewski (Ed.). Springer, 68–85.

[8] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, David Allen McAllester and Petri Myllymäki (Eds.). AUAI Press, 220–229.

[9] Manuel Kauers. 2013. The Holonomic Toolkit. In *Computer Algebra in Quantum Field Theory: Integration, Summation and Special Functions*, Carsten Schneider and Johannes Blümlein (Eds.). Springer, 119–144.

[10] Peter S. Maybeck. 1979. *Stochastic Models, Estimation, and Control*. Number 141 in Mathematics in Science and Engineering. Academic Press.

[11] Jeffrey W. Miller and Matthew T. Harrison. 2018. Mixture Models with a Prior on the Number of Components. *J. Amer. Statist. Assoc.* 113, 521 (2018), 340–356.

[12] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic Inference by Program Transformation in Hakaru (System Description). In *Proceedings of FLOPS 2016: 13th International Symposium on Functional and Logic Programming (Lecture Notes in Computer Science)*, Oleg Kiselyov and Andy King (Eds.). Springer, 62–79.

[13] Praveen Narayanan and Chung-chieh Shan. 2017. Symbolic Conditioning of Arrays in Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 11:1–11:25.

[14] David Lorge Parnas. 1976. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* SE-2, 1 (March 1976), 1–9.

[15] Judea Pearl. 1988. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

[16] Norman Ramsey and Avi Pfeffer. 2002. Stochastic Lambda Calculus and Monads of Probability Distributions. In *POPL '02: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 154–165.

[17] Chung-chieh Shan and Norman Ramsey. 2017. Exact Bayesian Inference by Symbolic Disintegration. In *POPL '17: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 130–144.

[18] Sam Staton. 2017. Commutative Semantics for Probabilistic Programming. In *Programming Languages and Systems: Proceedings of ESOP 2017, 26th European Symposium on Programming (Lecture Notes in Computer Science)*, Yang Hongseok (Ed.). Springer, 855–879.

[19] Robert Zinkov and Chung-chieh Shan. 2017. Composing Inference Algorithms as Program Transformations. In *Proceedings of the 33rd Conference on Uncertainty in Artificial Intelligence*, Gal Elidan, Kristian Kersting, and Alexander T. Ihler (Eds.). AUAI Press.