juliacon

Applied Measure Theory for Probabilistic Modeling

Chad Scherrer[1] and Moritz Schauer[2]

[1]Informative Prior
[2]Chalmers and Gothenburg University, Sweden
*Corresponding author: chad.scherrer@gmail.com

ABSTRACT

Probabilistic programming and statistical computing are vibrant areas in the development of the Julia programming language, but the underlying infrastructure dramatically predates recent developments. The goal of `MeasureTheory.jl` is to provide Julia with the right vocabulary and tools for these tasks.

In the package we introduce a well-chosen set of notions from the foundations of probability together with powerful combinators and transforms, giving a gentle introduction to the concepts in this article.

The task is foremost achieved by recognizing *measure* as the central object. This enables us to develop a proper concept of *densities* as objects relating measures with each others. As densities provide local perspective on measures, they are the key to efficient implementations.

The need to preserve this computationally so important locality leads to the new notion of *locally-dominated measure*, solving the so-called "base measure problem" and making work with densities and distributions in Julia easier and more flexible.

Keywords

Julia, Measure theory, Probability, Statistics

1. Why measures?

Distributions are an insufficient abstraction for probabilistic modeling.

Let's first consider Bayesian modeling. In the posterior density of the parameter θ given the observation or data x,

$$p(\theta \mid x) = \frac{p(\theta)\, p(x \mid \theta)}{p(x)} ,$$

the denominator is called the *model evidence*. Computing it efficiently is often difficult or unfeasible.

Some sampling algorithms circumvent this problem and only require knowing the density up to a constant factor. We then can work with the "unnormalized posterior density" instead, but we want to distinguish it from a proper probability density, because that difference determines which algorithms are available.

In developments like this using distributions, it's common to begin in terms of a distribution, but then carry around the unnormalized posterior as a function. Fortunately, the meaning of this function is usually clear from context. But structurally, this representation is now divorced from its meaning. It's no longer a Distribution object, so the tools from the original library can no longer help. This is a perspective we will take often.

Thus, starting with elements of our class of interest, a simple and common operation has led us to something outside of this class. This is analogous to the way polynomials over the reals lead to the complex numbers. If "the shortest path between two truths in the real domain passes through the complex domain" (Hadamard), then the same argument can be made for measures as connection between distributions.

As a second example, people working in Bayesian modeling sometimes use *improper priors*. Whatever one's position on the merits of this, we think it's reasonable to make this approach possible. But an improper prior does not integrate to one, so it's not a distribution.

There's also an elegant correspondence between frequentist and Bayesian methods, where regularization corresponds to a prior. But limiting ourselves to distributions restrains us from fully realizing this, because the corresponding "prior" might be improper or even flat, as in maximum likelihood estimation.

A final and very different concern is the structure of most distribution libraries, in which distributions are classified primarily according to whether they are "discrete" or "continuous". Such systems often lack facilities to use these in combination; some go so far as to encode the distinction in the type system, making them fundamentally incompatible.

But "discrete vs continuous" is a false dichotomy. For example, in three dimensions we can (and often do) work with distributions over points, lines, planes, or the entire space, or over spaces like a simplex or the surface of a sphere. These can be combined in rich ways, for example as a spike and slab prior for sparse modeling, or as a parameterized subspace for low-rank modeling.

We can address all of these points by extending the system we work with. Instead of distributions, our primary class of interest is *measures*, with distributions as a special case.

Of course, this special case is particularly useful. The point is not to disregard distributions, but to change our focus to one allowing a richer calculus for reasoning. Adapting Hadamard's argument, *The shortest path between two distributional truths passes through non-distributional measures*.

Contributions

Our work is novel in several ways. `MeasureTheory.jl` has

(1) Explicitly represented *base measures* with the same sophistication as the rest of the system, in particular more than just "discrete or continuous";
(2) A *local* approach for determining *absolute continuity*, which is usually a global characteristic;
(3) Multiple parameterizations for a given measure, without a proliferation of constructors; and
(4) Normalization and support constraints held separately from the data-dependent computation, allowing for greater efficiency.

Some parts of our approach can been seen in existing systems, though these are still far from universal:

- A rich set of *combinators* for building new measures from existing ones;
- Flexible type constraints, for example allowing measures with symbolic parameters;
- Light-weight measure construction, replacing a common assumption that once constructed, a measure will be used many times. This is especially important for probabilistic programming applications.

2. What are measures?

We'll now describe some foundations to help the reader get a deeper understanding of our approach, in particular what measures and probability distributions are and how they relate to notions such as the probability of something happening in the real world or the notion of volume of the space we are living in.

Throughout this discussion, it's important to keep in mind that for us the measure-theoretic abstractions are only means to an end.[1] Our primary interest, and the goal of `MeasureTheory`, is to offer support for *applied probabilistic modeling*. The package name is owed to the fact, that the word measure alone is too vague.

Given a space X of possible outcomes and a set of subsets of X called *events*, a probability distribution assigns each event a non-negative quantity, called the *probability* or *probability mass*. Likewise, a *measure* assigns each *measurable set* a non-negative quantity, also called a "mass".

The space X could be the space $\{1, 2, 3, 4, 5, 6\}$ suitable to model a six-sided die, or X might be the 3-dimensional Euclidean space $\mathbb{R}^3$ suitable to model "volume", to give two examples.

One further such space that deserves mention is traditionally denoted Ω. This is an abstract space connected to the real-world notion of *probability*, which we denote as Prob. Computationally, Ω can be considered to be the set of possible states of a random number generator `rng::AbstractRNG`, which is the source of computational randomness.

[1] The stackoverflow discussion `https://mathoverflow.net/q/11591` discusses books on measure theory; M.S. likes [22] as starting point and uses [10] and [2] as reference. As we anticipate interest in connecting with the actual theory, we give some pointers to more technical details in the footnotes.

Ω has a special role of tying random quantities a program produces and their distributions together, through the notion of random variables, which are functions $\mathbf{X}$, $\mathbf{Y}$ from Ω taking values in spaces X, Y such as $\{1, 2, 3, 4, 5, 6\}$ or $\mathbb{R}^3$.[2] The package `Omega.jl` [23] uses random variables derived from Ω as core principle.

From a computational perspective, a Julia function taking only an `rng::AbstractRNG` argument, making a number of calls to `rand(rng)`, and returning a value $x \in X$ is a random variable $\mathbf{X}\colon \Omega \to X$. A simple example is `X(rng=Random.GLOBAL_RNG) = rand(rng, 1:6)`. While mathematically $\mathbf{X}$ is a function from Ω, the argument ω is often hidden like the function argument `rng` of its computational counterpart—not a coincidence.

Each random variable $\mathbf{X}$ is tied to its *probability law* D, the distribution assigning probability to the events $\{\mathbf{X} \in A\}$ for each (measurable) set A

$$\mathrm{Prob}(\mathbf{X} \in A) = D(A).$$

In that sense, classical distribution packages restrict themselves very much to the task of providing a catalogue of useful probability laws, and a simple mechanism to provide a random draw (a random variable) with that law, as function `rand(rng, D)`. In the example,

$$\mathrm{Prob}(\mathbf{X} \in A) = D(A) = \frac{|A|}{6}$$

where $|A|$ denotes the number of elements of $A \subset \{1, 2, \cdots, 6\}$.

In `MeasureTheory`, measures have abstract super-type `AbstractMeasure`.

Kolmogorov's axioms

We now introduce *Kolmogorov's axioms*, which describe laws that characterize probability distributions, and, with one exception, also measures.

Measures and probability distributions are both required to obey the axiom that the (probability) mass of sets obtained as union of disjoint component sets[3], equals the sum of the (probability) masses of the components. So if sets A and B are *not* disjoint, the mass of the union is computed from the mass of the components using *inclusion-exclusion*. For probabilities this is

$$P(A \cup B) = P(A) + P(B) - P(A \cap B)\ .$$

Similarly for a measure μ,

$$\mu(A \cup B) = \mu(A) + \mu(B) - \mu(A \cap B)\ .$$

In the case where A and B are disjoint, this reduces to

$$P(A \cup B) = P(A) + P(B)\ ,$$

a property called *additivity*.[4]

[2] Additionally, one requires a random variable to be a *measurable function*, that is, one for which the inverse image of an event $\subset X$ is also an event of Ω. This is a similar, but much weaker requirement to the definition of a continuous function ("the inverse image of an open set is also open").

[3] Precisely: union of a sequence of disjoint component sets.

[4] The axioms also require a more general form of this to hold. If $\{A_n \mid n \in \mathbb{N}\}$ are *pairwise disjoint* (that is, no two have any common ele-

The reader has encountered many measures before, whether they were given that name. For example, *counting measure* gives the number of elements of a set, and the *Lebesgue measure* gives length, area, volume, etc in Euclidean space (`CountingMeasure` and `LebesgueMeasure` in the package.)

So far our characterizations of measures and probability distributions are functionally identical. Indeed, the one distinguishing feature is the *law of unit measure*, which only for distributions requires that

$$P(X) = 1 \,,$$

and also $\mathrm{Prob}(\Omega) = 1$. Thus Ω can be considered the event "that anything at all will happen".

By this axiom, probability mass is a proportion, a quantity between 0 and 1. Only this axiom, which for measures is replaced by weaker axiom $\mu(\varnothing) = 0$, sets the two apart. To reinforce that a distribution is also a measure, we'll sometimes refer to it as a *probability measure*.

This close relationship between measures and *probability* measures is illustrated by the Lebesgue measure λ on $X = \mathbb{R}$. The measure of the full space is $\lambda(\mathbb{R}) = \infty$. But restricting to the unit interval gives $\lambda([0,1]) = 1$. This *restricted* measure is a probability distribution—the uniform distribution $\mathrm{U}([0,1])$ describing the law of Julia function `rand()` giving random numbers in the interval $[0,1]$.

We have not yet stated *for which* sets Kolmogorov's axioms have to hold for something to be called a measure or a probability:

σ-algebras and how to like them

A (probability) measure on X need not assign a (probability) mass to every possible subset $A \subset X$, but only to each *event* for a probability measure, or *measurable set* more generally. Statements we make about sets should be understood as restricted to this set-of-sets, which can be different from one measure to the next.

Now, Kolmogorov's axioms require a given measure to be defined for X, and they also require that this set of measurable sets is closed under the operations which the axioms allow (complements and unions of sets or sequences of sets). This imposes an algebraic structure to the measurable sets, called a *σ-algebra*, which is exactly that: A set of sets $\subset X$ (or $\subset \Omega$ in particular) closed under complementation and countable unions and intersections.

This begs the question: Why not just use the *power set*, the "set of all sets", which is, after thinking about it, such a σ-algebra?

A key observation is that having defined probability mass for a number of relatively simple sets, for example on all intervals, Kolmogorov's axioms give a recipe to compute the probability mass of more complicated sets, and with those, even more in new applications of the axioms, etc. One says the σ-algebra is *generated* by these sets. Indeed, `Distributions.jl` does not give means to compute probability for arbitrary subsets of floating point numbers, but gives a way of computing probability mass just of intervals of the form $(-\infty, x]$ through the function `cdf`.

For spaces that are "small enough", a meaningful probability can be assigned to each subset of the space. Unfortunately, this is impossible for any space containing a continuum such as the real line. Just representing a single arbitrary subset of the `Float64`-range, the computational abstraction of the real line, would require staggering 2 306 Petabytes of memory. This is not practical.

But even the mathematical measure theory needs to stay clear of the set of all subsets of uncountable spaces, because it turns out that not even volume can be properly defined for all subsets of the Euclidean space.[5]

Accordingly, as of this writing, `MeasureTheory` does *not* have first-class σ-algebras, but rather considers them to be implicit to a given measure. Having said that, σ-algebras do have a role to play in applied measure theory, but we give only a pointer here. The practical importance of σ-algebras here lies therein that they also encode available information. Observe that each event A corresponds to a question, with a probabilistic answer. For example, an event $A = \{\mathbf{X} \in [c,d]\}$ corresponds to the question whether the random variable $\mathbf{X}$ is in the interval $[c,d]$ and there is a probability $\mathrm{Prob}(A)$ that this will be the case. Assume we know whether some of such questions A, B are answered (each with yes or no).

Then we also know that the answer to the complement A^c (no or yes), and we always know that Ω has probability 1 and the corresponding question is answered by 'yes'. Likewise, if both A and B can be answered given what we know, we also know the answer to $A \cup B$. That means, these known events form an σ-algebra, and in particular for each random variable $\mathbf{X}$ there is a σ-algebra associated with $\mathbf{X}$ which encodes what else we know knowing $\mathbf{X}$ and how we should assign (conditional) probabilities to other events knowing $\mathbf{X}$.

This is a very relevant question for probabilistic programming tasks and causal inference, and with *kernels* `MeasureTheory` does provide tools to reason about it which are computationally appropriate, because they are local.

Densities

A measure is often described in terms of its *density*. Before getting to a more technical discussion, we hope a physical analogy can help build some intuition. Imagine a wooden board with some knots in it, where we might be interested in the mass of some two-dimensional cut-out shape. This mass depends not only on the shape but also on its location and orientation, in particular the inclusion of knots. In this way we can start with a measure (here Lebesgue measure) and use a *density* (the physical density of the wood) to construct a new measure (the mass of any given cut-out shape).

Of course, we do this all the time with distributions, building a continuous distribution in terms of a *probability density function (pdf)* over Lebesgue measure, or a discrete distribution in terms

ments), then it must be true that

$$P\left(\bigcup_{n\in\mathbb{N}} A_n\right) = \sum_{n\in\mathbb{N}} P\left(A_n\right) \,,$$

and similarly for more general measures, replacing P with μ. This extension of additivity to the countably infinite case is called *σ-additivity*.

[5] Another thing to keep in mind is that, for a function $f\colon X \to Y$ to be measurable requires that "the inverse image of a measurable set must be measurable". So allowing more sets to $B \subset Y$ to be measurable requires either allowing more measurable $A \subset X$, or restricting the set of measurable functions.

of a *probability mass function (pmf)*, which is just a density over counting measure.

Somewhat more formally, a probability density in Euclidean space is a *local* ratio of probability assigned to an infinitesimally small volume/area $\mathrm{d}x$ around each point x, relative to that volume itself

$$f(x) = \frac{\text{Probability}(\mathrm{d}x)}{\text{Volume}(\mathrm{d}x)} .$$

"Volume" is not a distribution, but is in fact the Lebesgue measure described above. Discrete distributions can be expressed similarly using counting measure. There are certainly more events (sets) than outcomes (elements), so a probability density gives a local and parsimonious description of a distribution.

More generally, for measures μ and ν (and an *absolute continuity* condition, to be discussed), the density is

$$f(x) = \frac{\mu(\mathrm{d}x)}{\nu(\mathrm{d}x)} = \frac{\mathrm{d}\mu}{\mathrm{d}\nu}(x) .$$

As we see, density only makes sense relative to some *base measure*.

Limiting ourselves to distributions makes this awkward to even discuss, but allowing measures as first-class objects means we can make this characterization more explicit, and thus more flexible. What sets `MeasureTheory` apart from most libraries is that we don't sweep this under the rug, but rather address it head-on.

In place of Lebesgue or counting measure, any measure we can express can play the role of $\mathrm{d}\nu$ above. This becomes crucial in high-dimensional spaces, where working with other reference measures such as a product of normal distributions becomes a numerical requirement. (Mathematically, there is no infinite dimensional Lebesgue measure, so numerically, even high-dimensional Lebesgue measures can be problematic. [17] for example allows to express a target density with respect to product of normal distributions using the Boomerang sampler [4] for this reason.)

3. Locally dominated measures

One cannot expect to express a measure μ putting positive mass on a set S relative to a second measure ν on S, if nothing is there to compare, that is if $\mu(S) > 0$ but $\nu(S) = 0$. You can't "make something from nothing".

Given measures μ and ν defined on a common space X, we say μ is *dominated by* ν if $\nu(S) = 0$ implies $\mu(S) = 0$ (or equivalently, $\mu(S) > 0$ implies $\nu(S) > 0$) for every measurable S. This is denoted by $\mu \ll \nu$, and is sometimes equivalently read as "μ is *absolutely continuous* with respect to ν".

The Radon-Nikodym Theorem states that $\mu \ll \nu$ if equivalently there is a *density* function f, often written $\frac{d\mu}{d\nu} := f$, with the property that for every S,

$$\mu(S) = \int_S f \, \mathrm{d}\nu .$$

The concept of absolute continuity is very useful for formal manipulations. However, the global nature of testing $\mu \ll \nu$ would require every support to be represented in a way that allows efficient computation of this relation. In particular, in applications based on Markov chain Monte Carlo, such global information about the measures at hand is not available.

Even with such a capability, this approach would have its problems. It seems likely a user getting an error in response to requesting a density would often respond by restricting measures accordingly until the request can be fulfilled.

Because of this, we instead define μ to be *locally dominated* by ν near x, written $\mu \ll_x \nu$, if there is some neighborhood $N \ni x$ such that $\nu(N) > 0$ (so it's not a degenerate case) and $\mu|_N \ll \nu|_N$ (absolute continuity between the restricted measures). A *local density* can be defined similarly. In `MeasureTheory`, the `density` and `logdensity` functions work in exactly these terms. For the remainder of this paper, "(log-)density" will always refer to the local (log-)density, and "local" will often be taken as understood.

Density computations are typically done in log space. For example, if we're interested in $\frac{\mathrm{d}\mu}{\mathrm{d}\nu}$, we'll typically instead compute `logdensity(`μ`, `ν`, x)`. Here and below, we'll often write the math in terms of densities and the code in terms of log-densities, despite unfortunate inconsistency between the two.

There are several benefits to working in log-space. Results are much less likely to overflow or underflow, and the many products and exponents become sums and products, respectively. which are much more efficient to compute and to differentiate.

Working locally and in log-space also gives us a convenient *anti-symmetry*,

```
logdensity(μ, ν, x) == -logdensity(ν, μ, x)
```

In particular, `logdensity(`μ`, `ν`, x)` takes special values in some common cases:

	$\nu \ll_x \mu$	(else)
$\mu \ll_x \nu$	(finite)	`-Inf`
(else)	`Inf`	`NaN`

4. Base measures and density recursion

To this point, we've described the implementation in terms of the three-argument `logdensity(`μ`, `ν`, x)`. But implementing things directly in these terms would require a definition for every pair of possible measures, which is of course intractable.

We address this by instead assigning every measure a *base measure*. This is defined locally, with default method

```
basemeasure(μ::AbstractMeasure, x) = basemeasure(μ)
```

This allows users to define a base measure specific to a given neighborhood if they prefer, or to define a global base measure when that suffices.

The log-density is then fundamentally expressed in term of the two-argument `logdensity(`μ`, x)`, which gives the log-density with respect to this base measure. Thus we have

```
logdensity(μ, x) == logdensity(μ, basemeasure(μ, x), x)
```

This leaves us with the question of how to compute the three-argument `logdensity(`μ`, `ν`, x)` for general ν. We implement this recursively, in an approach we refer to as *density recursion*.

If α is the base measure of μ and β is the base measure of ν, then

$$\frac{\mathrm{d}\mu}{\mathrm{d}\nu} = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}\frac{\mathrm{d}\alpha}{\mathrm{d}\beta}\left(\frac{\mathrm{d}\nu}{\mathrm{d}\beta}\right)^{-1} .$$

In the implementation, `logdensity(`μ`, `α`, x)` and `logdensity(`ν`, `β`, x)` can be computed directly using the two-argument `logdensity`. There may be a specific method for `logdensity(`α`, `β`, x)`; if not, it will require a recursion step.

In this process, we may reach a measure that serves as its own base measure. Such measures are *primitive*, for example Lebesgue or counting measure. If `logdensity` is called on two primitive measures with no explicit method, an error is thrown.

The above serves as a high-level specification of the computational process. The low-level implementation will differ in some ways as we continue to tune it for performance.

5. Parameterized measures

A common challenge in building a library of distributions is the choice of *parameterizations*. For example, in Stan [7], a *negative binomial* distribution is parameterized by α and β, where

$$\mathrm{NegBinomial}(y \mid \alpha, \beta) = \binom{y+\alpha-1}{\alpha-1}\left(\frac{\beta}{\beta+1}\right)^{\alpha}\left(\frac{1}{\beta+1}\right)^{y} .$$

In the Julia package `Distributions.jl` [12], this is instead given by

$$\mathrm{NegBinomial}(y \mid r, p) = \binom{y+r-1}{r-1}p^{r}(1-p)^{y} .$$

These are equivalent (let $r = \alpha$ and $p = \frac{\beta}{\beta+1}$), and the inconsistency alone gives some evidence that it might be reasonable to prefer one or the other depending on the circumstances. Yet most libraries only allow one or the other (or yet another alternative) as *the* negative binomial distribution. Other parameterizations require entirely different names.

In `MeasureTheory`, our approach avoids this problem. A *parameterized measure* is defined by a struct of the appropriate type with a named tuple `par` field. For example,

```
struct NegativeBinomial{N,T} <: ParameterizedMeasure{N}
    par :: NamedTuple{N,T}
end
```

We can then write

```
NegativeBinomial(r=10, p=0.75)
```

or

```
NegativeBinomial(α=10, β=3)
```

Calls to `rand`, `logdensity`, etc then delegate to methods according to the appropriate names. We use `KeywordCalls.jl` [20], so all names are resolved statically at compile time.

6. Kernels

A *kernel* is a (measurable[6]) function κ that returns a measure. Equivalently, it represents a family of measures parameterized by its argument. Writing $\mathcal{M}(Y)$ for "measures on Y", we can write this as

$$\kappa\colon X \to \mathscr{M}(Y) .$$

A prominent application is a *conditional distribution*. In this case κ is further restricted to be a *Markov kernel*,

$$\kappa\colon X \to \mathscr{P}(Y) ,$$

where $\mathscr{P}(Y)$ represents "probability measures on Y". If $\mathbf{X}$ and $\mathbf{Y}$ are random variables defined on X and Y, the kernel κ defines a probability measure assigning every measurable $B \subset Y$ probability

$$P(\mathbf{Y} \in B \mid \mathbf{X} = x) =: (\kappa(x))(B) .$$

In `MeasureTheory`, a kernel is represented as either a Julia function or a `Kernel` object. The latter pairs a measure constructor with a mapping into its parameter space, and makes the functional relationship between argument of κ and the returned measure transparent.

A Markov kernel, in particular, corresponds to the distribution of a *parametrized random variable*, a function with random outcomes (or to "mechanisms" in causal inference, for example in [18]). For example in `MeasureTheory`, the kernel

$$\kappa\colon x \mapsto \mathcal{N}(x, \sqrt{x})$$

can be expressed as

```
κ = kernel(Normal) do x
    (μ=x, σ=√x)
end
```

or equivalently

```
κ = kernel(Normal; μ=identity, σ=sqrt)
```

corresponding to the random function

```
f(rng, x) = x + √x * randn(rng)
```

In this example, working in terms of a `kernel` allows us to make the linear relationship between the argument `x` and `mean(κ(x))` transparent.

This could of course also be expressed as a parameterized measure. The difference is that kernels are lighter weight to build and have all the dynamism of functions, while the static nature of parameterized measures can make it easier to express some optimizations.

Finally, kernels are key to expressing *likelihoods*, which we can use to formulate Bayesian conjugacy properties. We'll discuss this further in Section 9.

7. Markov chains

Some kernels κ map elements of a state space X to measures on the same space.

[6] That is, the function $x \mapsto \kappa(x)(A)$ must be measurable for every fixed A.

In this case, repeated application of the kernel "walks" through X, yielding the law of a *Markov chain*. We support this in `MeasureTheory` through the `Chain` combinator. This takes a kernel (or function) and a starting measure, and can be used with Julia's convenient do notation. For example, we can write a Gaussian random walk as

```
mc = Chain(Normal(μ=0.0)) do x Normal(μ=x) end
```

A random sample from a `Chain` is an infinite sequence expressed as a iterator, e.g.

```
julia> r = rand(mc);

julia> x = Iterators.take(r, 3) |> collect
3-element Vector{Any}:
 -0.4931543737034523
 -0.5661895116186417
 -1.3286977670590228
```

Thus, for example, we can evaluate `mc` on the `x` we just computed:

```
julia> logdensity(mc, x)
-0.4149771036439342
```

Crucially, the call to `rand` builds a struct containing a random seed, to ensure that repeated realizations of `r` will be identical.

To give us more control over the sampling procedure, `Chain` is implemented using *dynamic iterators* from `DynamicIterators.jl` [16].

The appropriate σ-algebra for infinite sequences requires some care. A Markov chain on X has σ-algebra generated by sets of *sequences with finitely many constraints*.[7]

8. Densities and integrals

In working with measures, two computations that come up quite often are (Radon-Nikodym) *differentiation* and (Lebesgue) *integration*. It's often important to be able to do things like

(1) For measures μ and ν, obtain $f = \frac{\mathrm{d}\mu}{\mathrm{d}\nu}$
(2) Given a measure ν and a function f, obtain a measure μ with $\mu(A) = \int_A f \, \mathrm{d}\nu$

Note that (1) is different from our current `density` function, because no x is specified. Computationally, this is called a *closure*.

In `MeasureTheory`, we write these as

$$\texttt{f = d}(\mu, \nu) \quad \text{and} \quad \mu = \textstyle\int(\texttt{f}, \nu)\,.$$

This allows the Radon-Nikodym theorem to be expressed as

$$\texttt{f == d}\textstyle\left(\int(\texttt{f}, \nu), \nu\right) \quad \text{and} \quad \mu == \int(\texttt{d}(\mu, \nu), \nu)\,.$$

Because we often we want to work instead in log-space, we also include (and tend to prefer) the functions $\texttt{logd}$ and $\int\texttt{exp}$, for which we would write

$$\ell = \texttt{logd}(\mu, \nu) \quad \text{and} \quad \mu = \textstyle\int\texttt{exp}(\ell, \nu)\,,$$

which follow a similar law,

$$\texttt{f == logd}\left(\textstyle\int\texttt{exp(f, }\nu\texttt{), }\nu\right) \quad \text{and} \quad \mu == \textstyle\int\texttt{exp(logd}(\mu, \nu\texttt{), }\nu\texttt{)}$$

Finally, in terms of `density` and `logdensity`, we have

$$\texttt{d}(\mu, \nu)(\texttt{x}) == \texttt{density}(\mu, \nu, \texttt{x})$$
$$\texttt{logd}(\mu, \nu)(\texttt{x}) == \texttt{logdensity}(\mu, \nu, \texttt{x})$$

9. Pointwise products and likelihoods

For a parameter θ and data x, suppose we have a *prior* $p(\theta)$ (momentarily assuming Lebesgue base measure) and *likelihood* $p(x|\theta)$. Then Bayes's Law gives the posterior density

$$p(\theta \mid x) = \frac{p(\theta)\, p(x \mid \theta)}{p(x)}\,.$$

In practice, we rarely know the normalization factor $p(x)$, so we often work in terms of the unnormalized posterior density,

$$p(\theta)\, p(x \mid \theta)\,.$$

Because this does not include the normalization, this does not integrate to one, and so it's not a probability density function. But there are no such problems as a density for a measure; the measure defined in this way has the same base measure as that of the prior.

This *pointwise product* in probability often describes the fusion of information, in this case the fusion of information from prior and from the observations via the likelihood, but also shows up in related situation, such as *sensor fusion* in signal processing.

More generally, this "prior times likelihood" can be considered as an operation that takes a measure and a function, and returns a new measure. Given a measure μ with base measure α, and a function f defined on the support of μ, $\mu \odot f$ is a measure with

$$\frac{\mathrm{d}(\mu \odot f)}{\mathrm{d}\alpha}(x) = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}(x)\, f(x)\,.$$

In the special case where f is a likelihood, it's important to keep track of whether it's represented in log space. For this, we have a `Likelihood` type that "knows" about the measure it was derived from and adapts according to whether the user calls `density` or `logdensity`. The general method

```
⊙(::AbstractMeasure, ::Likelihood)
```

defaults to constructing a `PointwiseProductMeasure` to hold the prior and likelihood, but we'll be extending this with more efficient methods for cases with conjugate priors.

In the absence of conjugacy, the extra structure of this suspended form is convenient, for example to build approximations to a given measure of interest.

The above `Likelihood` type is specified using a kernel and observed data. Though it is not an `AbstractMeasure`, we include a `logdensity` method for computational convenience. The core of the implementation is then

[7] That is, the smallest σ-algebra containing all sets of sequences which can be written as $\{x\colon x_i \in A_i \mid i \in I\}$ using collections of measurable sets $\{A_i \mid i \in I\}$ indexed by finite index sets I, called *cylinder sets.*

```
struct Likelihood{F,S,X}
    k::Kernel{F,S}
    x::X
end

function logdensity(ℓ::Likelihood, p)
    return logdensity(ℓ.k(p), ℓ.x)
end
```

10. Product and power measures

Given measures μ on X and ν on Y, the *(independent) product measure* $\mu \otimes \nu$ is a measure on $X \times Y$. Just as Lebesgue measure on $\mathbb{R}$ is *generated* by defining its value on intervals, the product measure is generated by defining

$$(\mu \otimes \nu)(A \times B) = \mu(A)\, \nu(B)\,,$$

for any measurable $A \subset X$ and $B \subset Y$. `MeasureTheory` code uses exactly this notation, `μ ⊗ ν`. This can be extended recursively to products of any finite number of measures.

If μ is defined in terms of a base measure α and likewise ν over β, the product measure has base measure $\alpha \otimes \beta$ and density

$$\frac{\mathrm{d}(\mu \otimes \nu)}{\mathrm{d}(\alpha \otimes \beta)} = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}\,\frac{\mathrm{d}\nu}{\mathrm{d}\beta}\,.$$

A special case of this is when $\mu = \nu$. In `MeasureTheory` we refer to this as a *power measure*, written `μ ^ 2` or `μ ^ n` for higher dimensions. More generally, the second argument can be a `Tuple`, so for example `μ ^ (2, 3)` extends μ as a product measure over 2×3 matrices.

A second special case is when we have a collection of values together with a kernel. For this we use the `For` combinator. So in a regression model, we might express the response as coming from

```
For(1:n) do j
    Normal(β * x[j], σ)
end
```

This is exactly the product measure

$$\bigotimes_{j=1}^{n} \mathcal{N}(\beta x_j,\, \sigma)\,.$$

More generally, suppose we have a measure μ on a space X and a kernel $\kappa\colon X \to \mathcal{M}(Y)$. Then we can define a measure on $X \times Y$ as follows.

For a given $x \in X$ with $\mu \ll_x \alpha$, let $\nu_x = \kappa(x)$ have base measure β_x. Then similarly to the independent product, we have

$$\frac{\mathrm{d}(\mu \otimes \nu_x)}{\mathrm{d}(\alpha \otimes \beta_x)} = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}\,\frac{\mathrm{d}\nu_x}{\mathrm{d}\beta_x}\,.$$

Because the types are always available to help us distinguish this from the independent case, it's safe to use the same notation, `μ ⊗ κ`.

Note that this defines a function

$$\otimes\colon \mathcal{M}(X) \times (X \to \mathcal{M}(Y)) \to \mathcal{M}(Y)\,.$$

In functional programming, this (together with some laws) is called a *monadic bind* [15].

11. Superposition and mixtures

A *superposition* is the measure-theoretic analog of a mixture model. For measures μ and ν with $f = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}$ and $g = \frac{\mathrm{d}\nu}{\mathrm{d}\beta}$, the superposition, written $\mu + \nu$, is a measure with base measure $\alpha + \beta$ and density

$$\begin{aligned}\frac{\mathrm{d}(\mu+\nu)}{\mathrm{d}(\alpha+\beta)} &= \frac{f\,\mathrm{d}\alpha + g\,\mathrm{d}\beta}{\mathrm{d}\alpha + \mathrm{d}\beta} \\ &= \frac{f\,\mathrm{d}\alpha}{\mathrm{d}\alpha+\mathrm{d}\beta} + \frac{g\,\mathrm{d}\beta}{\mathrm{d}\alpha+\mathrm{d}\beta} \\ &= \frac{f}{1+\frac{\mathrm{d}\beta}{\mathrm{d}\alpha}} + \frac{g}{\frac{\mathrm{d}\alpha}{\mathrm{d}\beta}+1} \\ &= \frac{f}{1+\left(\frac{\mathrm{d}\alpha}{\mathrm{d}\beta}\right)^{-1}} + \frac{g}{\frac{\mathrm{d}\alpha}{\mathrm{d}\beta}+1}\,.\end{aligned}$$

Using the inverse in this final line allows $\frac{\mathrm{d}\alpha}{\mathrm{d}\beta}$ to computed once and then re-used. Also note that in the special case where $\alpha = \beta$, this reduces to $\frac{f+g}{2}$.

An important special case of superposition occurs when μ and ν are finite measures on some space Ω, and $\mu(\Omega) + \nu(\Omega) = 1$. This is equivalent to a convex combination of probability densities, called a *mixture*.

Conveniently, a product of measures distributes over superposition. That is, if α and β are measures on a common space X and γ and δ are measures on Y, then

$$\alpha \otimes (\gamma + \delta) = \alpha \otimes \gamma + \alpha \otimes \delta\,,$$

and

$$(\alpha + \beta) \otimes \gamma = \alpha \otimes \gamma + \beta \otimes \gamma\,.$$

A very common special case of superposition is a *spike and slab prior*, a mixture of a *Dirac measure* (a point mass) with a continuous measure. This is useful for sparse Bayesian modeling, as implemented in the *sparse ZigZag sampler*, described in [5] and implemented in [17] using `MeasureTheory`.

12. Density decomposition

Especially for probability measures, it's common for the log-density with respect to Lebesgue or counting measure to have several types of terms. Given an observation x and any relevant parameters, there are typically

- Terms that are *data-dependent*, each involving some nontrivial function of x (and possibly also of the parameters),
- Terms that are *parameter-dependent*, and
- Terms that are *constant*.

In addition, it's common to have an "argument check" to be sure x is in the support of the distribution.

Depending on the application, we can often ignore some of these. For example, we may know x is in the support by construction, or from a previous check. In these cases, any use of resources to check arguments is wasteful.

There are many cases where it's important for performance to have constant and parameter-dependent terms separated from data-dependent ones. Suppose μ is a measure with log-density of the form

$$\log \frac{\mathrm{d}\mu}{\mathrm{d}\nu}(x) = \ell(x;\theta) = f(x,\theta) + g(\theta) + C \; .$$

If we instead observe an iid product of N observations, the log-density is

$$\begin{aligned}\sum_{j=1}^{N} \ell(x_j;\theta) &= \sum_{j=1}^{N} [f(x_j,\theta) + g(\theta) + C] \\ &= N[g(\theta) + C] + \sum_{j=1}^{N} f(x_j,\theta) \; .\end{aligned}$$

This final form can be much more efficient, because it reduces N repeated computations of $g(\theta) + C$ to one.

In some cases it's important to avoid computing $g(\theta) + C$ at all. For correlation matrices, it's common to use the LKJ prior [11].

Rather than work with correlation matrices directly, it's convenient to work in terms of the Cholesky decomposition. For this purpose, `MeasureTheory` includes an `LKJCholesky` measure. This is typically used as a prior with fixed parameters k and η, which give the dimensionality and "concentration" of the measure. The relative cost of normalization (which is irrelevant for MCMC) can be computed as

```
function relative_normcost(k, η)
    μ = LKJCholesky(k, η)
    L = rand(μ).L
    f_cost = @belapsed logdensity($μ, $L)
    g_plus_C_cost = @belapsed Dists.lkj_logc0($k, $η)
    return g_plus_C_cost / (g_plus_C_cost + f_cost)
end
```

So for example, in 10 dimensions with $\eta = 2.0$ this gives

```
julia> relative_normcost(10, 2.0)
0.76428528899935
```

That is, 76% of the time is spent in normalization. If our application doesn't need it, three-fourths of the computation time is simply wasted.

For these reasons, we break the representation of the log-density into several pieces (now additive terms in log-space):

(1) *Constant* terms, $-\frac{1}{2}\log 2\pi$.
(2) *Parameter-dependent* terms, $-\log\sigma$.
(3) *Data-dependent* terms, $-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2$.

The two-argument `logdensity` then computes only the data-dependent terms, with the constant and parameter-dependent terms pushed to the base measure. This makes it easy to defer computation of these terms until they are required.

13. Affine transforms

A particularly expressive way to build new measures in terms of existing ones is through a *pushforward*. Given a measure μ and a function f defined on its support, the *pushforward of μ through f* is a measure $f_*\mu$ defined by

$$f_*\mu(S) = \mu(f^{-1}(S)) \; .$$

In the following sections, we first discuss in the context of probability measures before the more general case.

Forward parameterization

Starting with a k-dimensional multivariate random variable z, an *affine transform* is a "linear transform with a shift". We can use this to define a new random variable x, as

$$\mathbf{X} = \sigma\mathbf{Z} + x_0 \; ,$$

with σ and x_0 of the appropriate dimensions.[8] If $\mathbb{E}[\mathbf{Z}] = 0$, then x has mean

$$\mathbb{E}[\mathbf{X}] = \sigma\mathbb{E}[\mathbf{Z}] + x_0 = x_0$$

and variance matrix

$$\mathbb{V}[\mathbf{X}] = \mathbb{E}[(\mathbf{X} - x_0)(\mathbf{X} - x_0)^t] = \mathbb{E}[\sigma\mathbf{Z}\mathbf{Z}^t\sigma^t] = \sigma\mathbb{V}[\mathbf{Z}]\sigma^t \; .$$

Note that if $\mathbb{V}[\mathbf{Z}] = I_k$, we get $\mathbb{V}[\mathbf{X}] = \sigma\sigma^t$, so we can arrive at a given positive semidefinite $\mathbb{V}[\mathbf{X}] = \Sigma$ by taking σ to be its *lower Cholesky factor*. Also, in the special case of a one-dimensional Gaussian, this gives the familiar $\mathbb{V}[\mathbf{X}] = \sigma^2$.

We call this the *forward parameterization* because it's especially convenient for sampling, sometimes referred to as "running the model forward". Unfortunately, the cost of this convenience is a relatively awkward expression for the density. In this case, we start with x and need to solve z, finally adjusting according to the determinant of the transformation:

$$p_{\mathbf{X}}(x) = \frac{1}{|\sigma|}\, p_{\mathbf{Z}}(z) = \frac{1}{|\sigma|}\, p_{\mathbf{Z}}\left(\sigma^{-1}(x - x_0)\right) \; ,$$

where $|\sigma|$ is the *determinant* of the square matrix, and the second equality comes from solving for z, which gives $z = \sigma^{-1}(x - x_0)$. More generally (when the transform is not expressed as a matrix), this role is played by the determinant of the *Jacobian*, $\left|\frac{\mathrm{d}x}{\mathrm{d}z}\right|$.

This requires solving a linear system. Even with σ being lower-triangular, this involves division operations and the allocation of a temporary vector for storage of z.

In many cases, we prefer the density (or log-density, really) to be fast to evaluate. This leads us to a kind of dual approach to the above.

Inverse parameterization

An alternative parameterization of a multivariate Gaussian is in terms of its *precision matrix*, $\Psi = \mathbb{V}[\mathbf{X}]^{-1}$. Similarly to above, we'll write ψ for the lower Cholesky factor of Ψ, so $\psi\psi^t = \Psi$. The parameterization is then specified by

$$\mathbf{Z} = \psi(\mathbf{X} - \mu) \; .$$

[8] We would typically use μ in place of x_0, if not for the unfortunate potential confusion with μ as a name for a measure.

Solving for z is of course now very simple, and the density becomes

$$p_{\mathbf{X}}(x) = |\psi|\, p_{\mathbf{Z}}(z) = |\psi|\, p_{\mathbf{Z}}(\psi(x-\mu))\ .$$

In exchange, forward sampling becomes awkward,

$$\mathbf{X} = \psi^{-1}\mathbf{Z} + \mu\ .$$

Generalization

Given an injective map $f\colon z \mapsto x$ and measures $\mu \ll_z \alpha$, the pushforward has density

$$\frac{\mathrm{d}\, f_*\mu}{\mathrm{d}\, f_*\alpha}(x) = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}\left(f^{-1}(x)\right) = \frac{\mathrm{d}\mu}{\mathrm{d}\alpha}(z)\ .$$

Note that there's no Jacobian to be found! This is because the measure and base measure are either both transformed, or both not. In fact, the Jacobian $\left|\frac{\mathrm{d}x}{\mathrm{d}z}\right|$ only comes into play when we compute "across the transform", and even then it's not in every case.

Changing our notation slightly, the general case is

$$\frac{f_*\mu(\mathrm{d}x)}{\alpha(\mathrm{d}z)} = \frac{1}{\left|\frac{f_*\alpha(\mathrm{d}x)}{\alpha(\mathrm{d}z)}\right|}\frac{\mu(\mathrm{d}z)}{\alpha(\mathrm{d}z)}\ .$$

This decomposes the problem into two subproblems. First we must compute $\left|\frac{f_*\alpha(\mathrm{d}x)}{\alpha(\mathrm{d}z)}\right|$. This plays the role of the determinant of the Jacobian, but is specific to the base measure α. In particular, α might be a discrete measure, in which case this factor is one. Finally, we compute $\frac{\mu(\mathrm{d}z)}{\alpha(\mathrm{d}z)}$, which is just the (pre-transformation) density, more familiar from previous discussion as $\frac{\mathrm{d}\mu}{\mathrm{d}\alpha}(z)$.

For the Lebesgue case, if the Jacobian $\left|\frac{\mathrm{d}x}{\mathrm{d}z}\right|$ is not square but, say, $n \times k$ for $n > k$, the resulting measure will be *embedded* into a k-dimensional affine subspace of $\mathbb{R}^n$. This can be convenient for low-rank modeling, which can be important for high-dimensional data. If σ has QR decomposition $\sigma = QR$, we can use $|R|$ in place of $|\sigma|$ or $\left|\frac{\mathrm{d}x}{\mathrm{d}z}\right|$ above, since columns of Q are orthonormal (so it's a change of basis and does not "stretch" the space). Our implementation is a variation of this that's equivalent but more efficient to compute.

14. Extensions

Despite it being a very new package, `MeasureTheory.jl` there is already active work to build up on it and to extend it.

`PointProcesses.jl`[8] defines *point processes*, in particular requiring the concept of *random measure*.

`ManifoldMeasures.jl`[1] implement measures on a *manifold*, using *Hausdorff measure* as the base measure.

`MultivariateMeasures.jl`[19] gives high-performance implementations of `logdensity` for multivariate measures, using `LoopVectorization.jl`[9].

`Soss.jl`[21] is a *probabilistic programming language* that has recently adopted `MeasureTheory.jl` as a foundation. In particular, every Soss `Model` is an instance of `AbstractMeasure`, and has another Soss model as its base measure.

As mentioned in Section 11, `ZigZagBoomerang.jl`[17] allows sampling with a spike and slab prior for sparse Bayesian inference and makes use the freedom to choose appropriate reference measures. These models can be expressed using Soss.

`Mitosis.jl`[24] uses `MeasureTheory.jl` to represent Bayesian networks via Markov kernels and defines transformations on those.

15. Related work

Narayanan et al [13] describe `Hakaru`, a system for Bayesian modeling using measures. Here, a measure is a functional

$$\mu[f] = \int f\ \mathrm{d}\mu\ ,$$

represented as a program. Hakaru's combinators are then expressed as compilers taking programs as the inputs.

Radul and Alexeev [14] describe the *base measure problem* of losing track of a base measure when applying a transformation, and suggest standardizing around Hausdorff measure as a solution. This problem doesn't arise for us, because the base measure is always taken into account.

Borgström et al [6] describes the `Fun` system in F# in terms of *measure transformer semantics*, but discusses only *finite* measures.

In Julia [3], the `Distributions.jl` package [12] is very popular for computations on distributions. The drawbacks of `Distributions.jl` are essentially those described in the first few sections of this paper. Current advantages over `MeasureTheory` are the extensive range of distributions it implements and its popularity and familiarity to many Julia users.

`MeasureTheory.jl` currently has `Distributions.jl` as a dependency, and uses it as a fall-back for many computations. For convenience, we also re-export the `Distributions` module under the handle `Dists`.

16. Conclusion

We have introduced the concepts and implementation of `MeasureTheory.jl`. This package is very new, so we expect there will be some changes as it matures. For this reason we have limited our discussion to aspects of the implementation we believe are relatively stable.

We hope this work can become a common foundation for probabilistic modeling in Julia. In particular, we believe this approach is especially well-suited for use in probabilistic programming, for which Julia has such a robust and active community.

We welcome discussion and community involvement with this package, as well as additional extensions to those we have described.

17. References

[1] Seth D. Axen. ManifoldMeasures.jl. `https://github.com/JuliaManifolds/ManifoldMeasures.jl`, 2021.

[2] Heinz Bauer. *Probability Theory and Elements of Measure Theory (PROBABILITY AND MATHEMATICAL STATISTICS)*. Academic Pr, hardcover edition, 1 1982.

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[4] Joris Bierkens, Sebastiano Grazzi, Kengo Kamatani, and Gareth Roberts. The Boomerang sampler. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 908–918. PMLR, 13–18 Jul 2020.

[5] Joris Bierkens, Sebastiano Grazzi, Frank van der Meulen, and Moritz Schauer. Sticky PDMP samplers for sparse and local inference problems, 2021. arXiv:2103.08478.

[6] Johannes Borgström, Andrew Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for bayesian machine learning. *Logical Methods in Computer Science*, 9(3), Sep 2013. doi:10.2168/lmcs-9(3:11)2013.

[7] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software, Articles*, 76(1):1–32, 2017. doi:10.18637/jss.v076.i01.

[8] Guillaume Dalle. PointProcesses.jl. `https://github.com/gdalle/PointProcesses.jl`, 2021.

[9] Christopher Elrod. LoopVectorization.jl. `https://github.com/JuliaSIMD/LoopVectorization.jl`, 2021.

[10] Jürgen Elstrodt. *Maß- und Integrationstheorie*. Springer Berlin Heidelberg, 2011. doi:10.1007/978-3-642-17905-1.

[11] Daniel Lewandowski, Dorota Kurowicka, and Harry Joe. Generating random correlation matrices based on vines and extended onion method. *Journal of Multivariate Analysis*, 100(9):1989–2001, 2009. doi:https://doi.org/10.1016/j.jmva.2009.04.008.

[12] Dahua Lin, John Myles White, Simon Byrne, Douglas Bates, Andreas Noack, John Pearson, Alex Arslan, Kevin Squire, David Anthoff, Theodore Papamarkou, Mathieu Besançon, Jan Drugowitsch, Moritz Schauer, and other contributors. JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions, Jul 2019. doi:10.5281/zenodo.2647458.

[13] Praveen Narayanan, Jacques Carette, Wren Romano, Chungchieh Shan, and Robert Zinkov. Probabilistic inference by program transformation in Hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 62–79. Springer, 2016. doi:10.1007/978-3-319-29604-3_5.

[14] Alexey Radul and Boris Alexeev. The base measure problem and its solution. *CoRR*, abs/2010.09647, 2020. arXiv:2010.09647.

[15] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–165, 2002.

[16] Moritz Schauer. DynamicIterators.jl. `https://github.com/mschauer/DynamicIterators.jl`, 2021.

[17] Moritz Schauer and Sebastiano Grazzi. mschauer/ZigZagBoomerang.jl: v0.10.0, 2021. doi:10.5281/ZENODO.3931118.

[18] Moritz Schauer and Martin Keller. mschauer/CausalInference.jl: v0.6.0, 2021. doi:10.5281/ZENODO.1005091.

[19] Chad Scherrer and Christopher Elrod. MultivariateMeasures.jl. `https://github.com/cscherrer/MultivariateMeasures.jl`, 2021.

[20] Chad Scherrer, Miles Lucas, Rafael Fourquet, and Simeon Schaub. KeywordCalls.jl. `https://github.com/cscherrer/KeywordCalls.jl`, 2021.

[21] Chad Scherrer and Taine Zhao. Soss: Declarative probabilistic programming via runtime code generation. Zenodo, Oct 2020. doi:10.5281/zenodo.5520061.

[22] A. N. Shiryaev. *Probability*. Springer New York, 1996. doi:10.1007/978-1-4757-2539-1.

[23] Zenna Tavares, James Koppel, Xin Zhang, and Armando Solar-Lezama. A language for counterfactual generative models, 2019.

[24] Frank van der Meulen and Moritz Schauer. Automatic backward filtering forward guiding for Markov processes and graphical models, 2020. arXiv:2010.03509.