A combinator library for MCMC sampling

Praveen Narayanan
Indiana University

Chung-chieh Shan
Indiana University

The current research and eventual goal of probabilistic programming focuses on reusing and composing models. However, domain experts and machine-learning experts need to work together not only on realistic models but also on efficient inference. The latter need leads us to ask how to apply programming-language facilities for modularity to reuse and compose inference techniques as well. For MCMC sampling in particular, we want to reuse and compose proposal distributions, transition kernels, and operations on sample-streams.

A popular approach to enable reuse and composition is to build a combinator library, which can be regarded as a domain-specific language embedded in a general-purpose language. Adopting this approach, we have built a combinator library for MCMC sampling.

The main goal of our library is to make it easy for domain experts to express their knowledge about inference while reusing and composing existing components. Accordingly, the main feature that distinguishes our library from other probabilistic programming systems is that a user can define custom proposal distributions, transition kernels, and operations on sample-streams. To this end, it is especially helpful that our library can be regarded as a domain-specific language embedded in the general-purpose language Haskell, so these custom definitions can use all Haskell facilities. The current version of the library can be found in the Haskell package mcmc-samplers [6].

1 Basic distribution types

Our library includes types and combinators for proposal distributions, transition kernels, and sample-stream processors. A typical MCMC method, such as Metropolis-Hastings sampling, relies on the ability

- to compute the density at a point of the target distribution,
- to compute the density at a point of the proposal distribution, and
- to sample from the proposal distribution.

Accordingly, we start by defining types for probability densities, sampling procedures, and steps in a random walk:

```haskell
type Density a = a → Double

type Sample a = Rand → IO a

type Step a = Rand → a → IO a
```

Here, \texttt{Rand} is a source of randomness (such as a PRNG) and the type \texttt{Sample} is to be read as a verb, i.e., “to sample”. Thus, a value of type \texttt{Density a} is a function that takes an input (of type \texttt{a}) and returns a probability density (of type \texttt{Double}). A value of type \texttt{Sample a} is a function that takes a source of randomness (of type \texttt{Rand}) and returns an action producing a sample (of type \texttt{a}). A value of type \texttt{Step a} is a function that takes a source of randomness and a current state and returns an action producing a next state.

We then define the types \texttt{Target a} for target distributions and \texttt{Proposal a} for proposal distributions. Whereas a target distribution is only required to provide a probability density, a proposal distribution is required to provide not only a probability density but also a
sampling procedure. To allow users of our library to construct their own proposals and
targets from scratch, the library provides two combinators with the following types:

\[
\text{makeProposal :: Density } a \rightarrow \text{ Sample } a \rightarrow \text{ Proposal } a \\
\text{makeTarget :: Density } a \rightarrow \text{ Target } a
\]

Thus, the function makeProposal provided by the library constructs a Proposal from a
Density and a Sample, whereas the function makeTarget constructs a Target from a
Density alone. However, most proposals and targets are constructed not from scratch
but using other combinators. To start with, the library defines (the probability densities
and sampling procedures of) standard proposal distributions, including

\[
\text{uniform :: Double } \rightarrow \text{ Double } \rightarrow \text{ Proposal Double} \\
\text{normal :: Double } \rightarrow \text{ Double } \rightarrow \text{ Proposal Double} \\
\text{bern :: Double } \rightarrow \text{ Proposal Bool}
\]

and more advanced primitives. The library also provides combinators for mixing these
proposals and focusing them on parts of the MCMC state.

2 Inference combinators

To explain these combinators in action, we consider a simple Gaussian mixture model
(GMM). Suppose we have a bunch of i.i.d. observations sampled from an unknown mixture
of two one-dimensional Gaussian distributions. To infer the mixture parameter, Gaussian
parameters, and the observation labels, we can write the following code. First we define the
MCMC state as a record data type in Haskell. The four fields of the record are
labels (a list of observation labels), gaussParams (the parameters of the two Gaussians), bernParam
(the mixture proportion), and obs (the observed data).

\[
\text{data GaussianMixtureState = GMM} \\
\text{\{ labels :: [Bool], gaussParams :: ((Double, Double), (Double, Double))} \\
\text{\, bernParam :: Double, \quad obs :: [Double] \}}
\]

Square brackets [] above denote lists. The length of these lists is the number of observed
data points, nPoints.

nPoints :: Int

We now define gmmProposal, our GMM proposal distribution, as a mixture of three distrib-
utions.

\[
\text{gmmProposal :: GaussianMixtureState } \rightarrow \text{ Proposal GaussianMixtureState} \\
\text{gmmProposal gmm = mixProposals} \\
\text{[ (updateLabels labelsProposal gmm, 10) } \\
\text{\, (updateGaussParams gaussParamsProposal gmm, 1) } \\
\text{\, (updateBernParam bernParamProposal gmm, 2) ]}
\]

This definition uses the mixProposals combinator in the library to mix the three compo-
nent proposals. This combinator turns a list of proposal-proportion pairs into a combined
proposal, while defining the correct mixture Density and Sample procedures for us.

mixProposals :: [(Proposal a, Double)] \rightarrow Proposal a

Without the mixProposals combinator, we would have to write both the Density and
Sample specific to this mixture. The mixture Density needs to compute a weighted sum
of the component densities, and the mixture Sample involves sampling from the categorical
distribution over the proposal-proportion pairs. The combinator does this work for us over
the abstract type Proposal a, which omits the need to redefine these procedures when we
make changes to the mixture components.
The three component proposals each affect a different part of the MCMC state. For example, 
labelsProposal affects the labels part of the GaussianMixtureState. As the type below 
shows, labelsProposal is focused on the type [Bool] (that is, a list of labels) rather than 
the entire GaussianMixtureState.

labelsProposal :: [Bool] → Proposal [Bool]

In the definition of gmmProposal above, we use the focusing combinator updateLabels, a 
function from functions to functions, to convert labelsProposal into a proposal for the 
entire GaussianMixtureState. (We similarly handle the other two component proposals, 
gaussParamsProposal and bernParamProposal.)

updateLabels :: ([Bool] → Proposal [Bool]) → 
GaussianMixtureState → Proposal GaussianMixtureState

Currently, the user needs to write boilerplate code for focusing combinators such as 
updateLabels. We plan to provide a code generator to produce this boilerplate code automatically.

To define the component proposal labelsProposal—

labelsProposal :: [Bool] → Proposal [Bool]
labelsProposal ls = chooseProposal nPoints f 
  where f n = updateNth n flipBool ls
    flipBool bn = if bn then bern 0 else bern 1

—we in turn use the uniform mixing combinator chooseProposal and the focusing combi- 
nator updateNth provided by the library.

chooseProposal :: Int → (Int → Proposal a) → Proposal a
updateNth :: Int → (a → Proposal a) → [a] → Proposal [a]

We define the target distribution gmmTarget using a combinator that joins probability den- 
sities without sampling procedures.

gmmTarget :: Target GaussianMixtureState 
gmmTarget = makeTarget (combineDensities 
  [labelsTarget, gaussParamsTarget, bernParamTarget, obsTarget])

Finally, we use the metropolisHastings combinator to define a transition kernel, which 
samples from gmmProposal and uses the probability densities to compute the Metropolis- 
Hastings acceptance ratio.

gmmMH :: Step GaussianMixtureState 
gmmMH = metropolisHastings gmmTarget gmmProposal

If we want, we could use other combinators in the library to construct Steps. For example, 
we can construct a mixture of Steps using mixSteps, which is analogous to mixProposals 
above.

mixSteps :: [(Step x, Double)] → Step x

Another combinator we could use is cycleSteps. Alternatively, if we want the mode of the 
target distribution, we can replace metropolisHastings by simulatedAnnealing.

As is, the transition kernel gmmMH can be invoked repeatedly, and the resulting stream of 
samples fed to a variety of processors that execute an action at each step of the random 
walk.

run = do
  rng ← createSystemRandom
  let state0 = ...
  walk gmmMH state0 (10^-6) rng (every 100 display)
Here we invoke the kernel one million times and display every 100th sample, using the 
walk, display and every combinator to do so. The library defines other sample-stream 
operations such as collect and batchPrint, and provides facilities to build more custom 
stream processors.

3 Related, current, and future work

Given that we emphasize custom and composable inference, the related works that are 
currently most relevant to us are Venture’s inference language [3] and FACTORIE’s infra-
structure for MCMC inference [5]. Other systems as far as we are aware (such as Church 
[1], BLOG [2], STAN [8], MCMCpack [4]) limit the user to choosing from a finite set of 
transition kernels and proposal distributions, whereas our library allows an infinite number 
of combinations (including mixing and cycling) and custom definitions from scratch.

We are currently extending the library to support data-parallel inference [7].
We will soon compare samplers written using our library against handwritten samplers. We 
expect our samplers to turn out to be much more concise and reusable, and not much slower.

Currently, the transition kernels in our library recompute the target densities in their entirety 
in order to calculate the acceptance ratio, even though most factors in the densities may 
cancel each other out in the ratio. Exactly which factors cancel out depends on the focus of 
the proposal sampled. A handwritten transition kernel would avoid computing the factors 
that cancel out. We could use our mixSteps combinator to express the same optimization, 
but the user currently needs to figure out by hand which factors cancel out. We want to track 
dependencies in the probabilistic program and perform this optimization automatically.

For the future, we plan to add more sampling methods to the library, such as Gibbs, HMC 
and reversible jump. Already those methods can be defined from scratch (or using the 
existing combinators) by the user each time they are desired, but we want the library to 
provide them once and for all as reusable and composable blocks.

References

[1] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and 
Joshua B. Tenenbaum. Church: a language for generative models. CoRR, abs/1206.3255, 
2012.
[2] Lei Li and Stuart J. Russell. The blog language reference, Technical Report UCB/EECS-
2013-51, EECS Department, University of California, Berkeley, May 2013.
[3] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order prob-
abilistic programming platform with programmable inference. CoRR, abs/1404.0099, 
2014.
gramming via imperatively defined factor graphs. In Neural Information Processing 
Version 2.5.0, 2014.