My research supports and enhances human creativity through computation. Specifically, I focus on solving complex procedural modeling and design problems, where content creators write programs that pseudo-randomly generate visual content, such as 3D models, web pages, or arrangements of furniture. I tackle these problems using probabilistic programming, a class of general-purpose tools for probabilistic modeling and inference. My work is interdisciplinary, leveraging insights from computer graphics, artificial intelligence, and programming languages. I develop composable, general-purpose solutions, rather than specialized fixes for one problem or application domain. This means that on the way to solving modeling and design problems, my work often leads to systems and algorithms that have broader impact for artificial intelligence and machine learning in general.

The procedural approach to content creation allows designers to rapidly create and explore alternative design possibilities. It also scales to creating large amounts of diverse content without extensive human effort, which is becoming more important as the virtual worlds of films, games, and simulations grow larger and more detailed. Traditional procedural modeling tools operate via forward reasoning: the user writes a step-by-step recipe for generating content, such as a 3D model for a house. However, detailed, realistic content must also satisfy properties that can be difficult for the programmer to guarantee constructively—for example, the house models should be structurally stable. Satisfying such conditions seems to require inverse reasoning, working backwards from the desired result.

Bayesian probabilistic inference provides a well-studied mechanism for this kind of inversion: the forward recipe serves as a generative prior, and the desired properties can be encouraged via a likelihood function. One way to pose and solve Bayesian inference problems is through probabilistic programming. Probabilistic programming languages are Turing-complete, universal languages augmented with random choices, conditioning and likelihood statements, and general-purpose inference operators. This universality makes them well-suited for representing procedural modeling and design problems, as they can capture the highly structured, often recursive, mixed-continuous-discrete nature of such problems. They are also human-readable and human-writable and require little-to-no special expertise in probabilistic inference to author or understand.

Unfortunately, the expressiveness of probabilistic programs comes at a cost: inference on such complex programs is often intractable. We must resort to approximate methods that converge slowly, if at all. Thus, most of my work has focused on improving the reliability and runtime efficiency of probabilistic programming inference, with a particular eye towards procedural modeling and design applications. My research in this space has involved both algorithmic work (increasing the statistical efficiency of inference through new or improved algorithms) as well as systems work (making existing inference algorithms more computationally efficient).

The ‘Lightweight MH’ variant of the Metropolis Hastings algorithm is a popular technique for implementing probabilistic programming inference. It is easy to implement and can be embedded in an existing deterministic language, rather than requiring its own specialized interpreter or compiler. And
it operates on arbitrary programs, including the complex programs found in procedural modeling and
design applications. Unfortunately, it is also inefficient: every Metropolis Hastings proposal triggers
a complete re-execution of the program, despite the fact that the proposal often only changes a small
subset of the program’s execution trace. Probabilistic languages that run on custom interpreters often
exploit this fact for faster inference, but they are complicated to implement and do not integrate as
readily with existing deterministic code.

To get the best of both worlds—efficient, incrementalized proposals in a lightweight, embedded
system—I developed a new MH implementation technique called C3 [9]. C3 leverages two insights.
First, it transforms programs into continuation passing style (CPS): this makes continuations available
at every point in the program, so execution can resume from anywhere. Second, C3 caches the receiver,
inputs, and output of each function call, since many calls can be skipped when these quantities have
not changed.

C3 significantly accelerates MH proposals. In the procedural modeling example below, where a tree-
generating program is encouraged to take the target shape on the left, C3 runs nearly 10 times faster
than Lightweight MH. Its two components, caching and CPS, work synergistically, delivering greater
speedups when used together than the sum of their individual contributions. C3 also works well for
many latent variable models in Bayesian data analysis, delivering 20-100x speedups on hidden Markov
models, Gaussian mixtures, topic models, and hierarchical regression models. C3 is publically avail-
able as the IncrementalMH inference method in the open-source WebPPL programming language [3].

DEALING WITH TIGHT CONSTRAINTS USING HMC

Lightweight MH is a ‘single-site’ Metropolis Hastings algorithm: it proposes changes to one random
variable at a time. This strategy can break down when faced with likelihoods that induce tight con-
straints between multiple variables. For example, constraining the interior volume of a procedurally-
generated bottle couples every random variable in the program. Changing one variable without also
performing a corresponding change to the others leads to a change in volume and thus a drop in
likelihood, so single-site MH struggles in these situations. Some programs admit problem-specific
multi-variable proposal strategies to get around this issue: in the bottle example, cross-sections of the
shape could be carefully adjusted in pairs to preserve volume. But these strategies can be difficult to
discover and design, and they do not generalize to new problems.

In my work, I recommend using Hamiltonian Monte Carlo (HMC) in such situations instead [7]. HMC
simultaneously proposes changes to multiple continuous variables; these proposals stochastically fol-
low the probability gradient, increasing the chance that they will result in a high-probability state. The
necessary gradient can be derived computationally using automatic differentiation, so it need not be
written explicitly by the programmer. These properties make HMC an attractive, general-purpose
candidate for inference in tightly-constrained procedural models.
I have shown that HMC successfully generates design suggestions in two different domains: constrained vector art coloring and stable stacking structures. The figure below shows several stable variations on a complex arch-like structure that HMC discovered, despite the tight global constraint introduced by encouraging structures to be in static equilibrium (Left). By contrast, single-site MH quickly becomes stuck in the first stable configuration it finds (Right). The source code for these examples is available online, and the underlying HMC implementation is part of Quicksand, my open-source system for high-performance probabilistic programming [6]. My colleagues and I have also physically fabricated some of the structures generated by this method. Given the success of this work and the increasing interest in computational fabrication, I believe there are more exciting possibilities at the intersection of procedural modeling and manufacturing.

HMC provides efficient inference over continuous variables, but modeling and design programs also often feature discrete, structural variables: for example, a procedural tree model has random choices that dictate when to generate new branches. When constrained via a likelihood function, such programs can exhibit “structural local maxima”: local maxima of the likelihood function that MH methods can only escape with carefully-coordinated proposals involving structural variables. The discrete, combinatorial nature of this problem makes it especially difficult to solve. There exist modifications to MH which can reliably escape such local maxima, such as the locally-annealed reversible jump algorithm, but they can require significant extra computation to do so [11].

Instead, I have proposed using Sequential Monte Carlo (SMC) for inference on such programs [8]. SMC generates a set of samples, called particles, in parallel. At regular intervals during the generation process, particles are resampled according to their likelihood score thus far: high-scoring particles tend to be duplicated, while low-scoring ones tend to be killed off. By receiving likelihood feedback throughout generation, SMC can steer its samples away from structural local maxima before they finish generating.

SMC was originally developed for time series models—to use it with programs that feature structured branching, the function call trees of those programs must be linearized. My work has shown that SMC’s performance is sensitive to the linearization order chosen, and that randomizing the order of each particle leads to stable high performance. This strategy is easy to implement using a simple stochastic variant of the ‘future’ parallel programming primitive. The resulting algorithm is called Stochastically Ordered Sequential Monte Carlo (SOSMC).

For recursively branching procedural models, SOSMC often converges significantly faster than MH. For example, in the figure below, SOSMC generates a spaceship model that closely matches the target shape after less than a second. In contrast, MH takes longer to converge to a matching result, since it must spend time escaping the structural local maximum into which it randomly initializes. An interactive demo comparing SOSMC to MH can be found online at http://dritchie.github.io/web-procmod.
SOSMC also composes usefully with the other systems and algorithms I have described. In a process called *particle rejuvenation*, either C₃ or HMC can be used to randomly diversify a set of partially-generated SOSMC particles, generating more varied design suggestions. Ultimately, I envision all of these methods as interchangeable tools in a single programming ecosystem. In fact, the WebPPL language already supports all of them and is rapidly improving their interoperability.

**FUTURE RESEARCH AGENDA**

Probabilistic programming for procedural modeling and design is a research field in its infancy, and there are many directions of future work to explore. First, though we have made major strides in improving the performance of probabilistic programming inference, there are compelling applications that require it to run even faster: games must generate new procedural content on the fly; interactive design tools must react in real time to user input. I plan to address these challenges through *amortized inference*, investing in pre-computation so that online inference runs faster [1]. Recent work in variational inference is particularly exciting: given a generative model of data, deep neural networks can be used to build and train a ‘recognition model’ that inverts the generative process, taking the data as input and sampling values of the latent variables [5]. These are promising methods, but further work is needed to adapt them for the structure-changing programs and non-data-driven likelihoods used in procedural modeling and design.

I also plan to examine how probabilistic programs are written to begin with. My inference work has assumed the existence of a complete program, but programs do not materialize fully-formed: people must write them piece by piece. Some pieces might be easily specified with code, such as the program’s overall control flow structure. But others might be more naturally expressed through input/output examples, such as the arithmetic computations that orient and place parts of a 3D model. I envision a continuum between programs written by hand and those induced from examples, with mixed-initiative programming tools in the middle that allow users to switch freely between these styles. Building and evaluating such tools will require extensive user studies and controlled experiments.

My future work will also move in the related direction of viewing *inference* as an interactive process. Systems I have built thus far have cast inference as an offline operation: the user provides a program, the system performs inference to generate results, the end. But design is inherently iterative: results are evaluated, and that evaluation feeds back into redesigns. To fit into this framework, inference must also react to feedback. One way to achieve this goal is to build mixed-initiative design tools, where the system suggests designs, the user selects—potentially combining and modifying—the results she likes best, and then subsequent invocations of inference condition on this feedback. I plan to investigate programming paradigms that make such feedback-reactive tools easy to author.

Finally, I plan to explore connections between my work in procedural modeling and other related research fields. For example, computer vision has seen a recent resurgence in ‘vision as inverse graphics’
systems, where a generative prior proposes possible 3D scenes, and a likelihood function scores how well those scenes match an observed image [4]. Structured procedural models might prove to be effective priors for these problems, and the inference methods I have developed could help solve them. I plan to apply similar techniques to 3D reconstruction, where the observed data includes depth images or point clouds, rather than just a single color image.

REFERENCES


