PROBABILISTIC PROGRAMMING FOR PROCEDURAL MODELING AND DESIGN

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Daniel Ritchie
August 2016

This dissertation is online at: http://purl.stanford.edu/vh730bw6700

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Pat Hanrahan, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Noah Goodman, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Procedural modeling, or the use of pseudo-random programs to generate visual content, plays an important role in computer graphics and design. It facilitates content creation at massive scales, can automatically create painstakingly-detailed imagery, and can even generate pleasantly surprising results that can help people to navigate large and unintuitive design spaces.

Many procedural modeling applications demand *directable* content—the ability to enforce aesthetic or functional constraints on model output. Bayesian inference provides a framework for such control: the procedural model specifies a generative prior, with the constraints encoded as a likelihood function.

In this dissertation, we use probabilistic programming languages to express such Bayesian procedural models. A probabilistic programming language (PPL) provides random choice and Bayesian conditioning operators as primitives, and inference corresponds to searching the space of program executions for high-probability execution traces. PPLs can concisely express arbitrary probability distributions, including the complex, hierarchical, and often recursive stochastic processes required in procedural modeling.

Unfortunately, PPL inference is typically inefficient, relying on expensive Monte Carlo methods. This dissertation develops a suite a of techniques for speeding up PPL inference. First, we eliminate redundant computation from Metropolis-Hastings, one of the most commonly used PPL inference algorithms. Next, we show how to efficiently explore tightly-constrained design spaces using the gradient-based, Hamiltonian Monte Carlo algorithm. Then, we address the problems posed by deep branching

structures in procedural models such as recursive trees, using a new variant of Sequential Monte Carlo. Finally, we make procedural modeling programs learn from their own outputs, using neural networks that guide programs toward high-probability results with fewer samples. Together, these tools bring PPL inference closer to interactive rates, which can enable more compelling graphics and design applications.

Acknowledgement

"How you get there is the worthier part."

I would not be here today, writing this dissertation, without the encouragement and support of many, many people.

I have to begin by thanking my first computer science instructors at Berkeley for kindling my interest in the field: Brian Harvey, for helping me believe that I could be a computer scientist (and not to be scared of the math); Paul Hilfinger, for giving challenging assignments that forced me to learn how to ask for help; Dan Garcia, for his bottomless enthusiasm, demonstrating that passion and computing are far from mutually exclusive. Finally, James O'Brien, for giving me my first opportunities to try my hand at research. I would not have made it to Stanford without those early experiences under my belt.

For funding my early years at Stanford, I thank Dr. Vishal I. Sikka of SAP. His Stanford Graduate Fellowship gave me enormous freedom to explore my research interests and to settle into a niche where I could be productive and successful. For my later years at Stanford, I thank DARPA for generously and open-mindedly supporting the fledgling field of probabilistic programming.

I have been blessed with extraordinary mentors while at Stanford—first and foremost, my adviser, Pat Hanrahan. At the beginning of my graduate studies, Pat was patient with me, encouraging me to pursue my own ideas and to take ownership of them. As I found my footing, he has become an incredible source of wisdom on how to shape a research project into a coherent and compelling story. Throughout, he has

run an eclectic, happy, and supportive research group of which I've been proud to be a part. He makes it all seem effortless, when it must be anything but. I don't know how he does it. I also owe a tremendous debt of gratitude to my co-adviser, Noah Goodman. As my interests turned more toward artificial intelligence (and probabilistic programming, in particular), Noah has been an invaluable source of technical expertise. I always leave my meetings with him feeling charged-up about the exciting possibilities for my research. I am also thankful to Tom Funkhouser for coaching me through my first paper rejection and for teaching me the basic approach to paper writing that I still use to this day.

My fellow students and colleagues have also been a continual source of inspiration, support, and encouragement. Thanks first of all to Jerry Talton and Yi-Ting Yeh: without your examples to follow, I would never have set off on this crazy mission of using probabilistic programs for design. The hugest of thanks goes to the members, past and current, of Pat's research group that I've gotten to know—Hanrahats forever! I am especially grateful to Matt Fisher, Manolis Savva, Sharon Lin, and Zach Devito, with whom I've collaborated over the years and who have taught me virtually everything I know about the day-to-day process of doing research. I also want to thank the wonderful people who have shared Gates 381 with me during my stay: Matt, Manolis, Angel Chang, James Hegarty, Michael Mara, and (last but certainly not least) the incomparable Katherine Breeden. You've made it a joy to come in to the office, even on those days when none of my research was working. I've been lucky to get to work with brilliant and insightful people from Noah's group, as well—thanks to Andreas Stuhlmüller, Siddharth Narayanaswamy, and Paul Horsfall for sharing your knowledge and broadening my horizons. Finally, thanks to the amazing undergraduates I've had the privilege to work with while at Stanford, Ben Mildenhall and Anna Thomas. It's almost scary how smart you are.

Outside of research, I want to thank my friends from the Sci-Fi and Fantasy Book Club at Stanford. The conversations, debates, and general nerd-out sessions we've shared together have added a joyous new dimension to my life that I didn't know I was missing. I never imagined I could look forward to Monday nights so much. And I cannot possibly express in words how grateful I am to the community of friends

I found through social dance at Stanford. From Jammix to Friday Night Waltz to Opening Committee and beyond, there are simply too many of you to name here. You have been my home away from home for the past six years; you've kept me sane, kept me laughing, and kept reminding me that my worth as a human being extends far beyond my output as a researcher. I have lived a fuller life because of you, and I can never thank you enough for that.

Finally, and most profoundly, thanks to my parents, Chris and Martha Ritchie, for their boundless love and support. You are and have always been my best friends and my strongest advocates. You've pushed me to be my best, but you've also encouraged me to do what makes me truly happy—I think most children would be lucky to get even one of those from their parents. I have to steal a line from one of my favorite movies, here: Mom, Dad—everything good in my life is because of you. I love you, and I still want to be like you when I grow up.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Randomness plays a key role in creative design. Visually interesting texture often appears random—just ask Jackson Pollock. Randomness can also create variations on a single design. For example, customers of the design studio Nervous System can personalize their own random variants of the studio's Kinematics jewelry [110]. And there exists evidence that the biggest creative innovations and breakthroughs happen by exploring random tangents which lead to serendipitous discoveries, rather than single-minded pursuit of a driving objective [105].

In computer graphics, the act of using randomness to create visual content is often called *procedural modeling* [24]. Procedural models are computer programs that pseudo-randomly generate some visual artifact. They can add naturalistic details to virtual scenes, such as the bark texture on a virtual tree or ripples on virtual water. They can create a dizzying variety of content at very little cost, such as the endless blocky environments created by Minecraft's world generator. They can also operate interactively, generating suggestions for a human user to approve. For example, the FaceGen software used to create player facial appearance in many video games can create faces at random, possibly arriving at interesting designs the player would not have thought of independently [44].

In many cases, it is desirable for these programs to be 'directable,' meaning it should be possible to control their output to satisfy specific criteria. These criteria

might be aesthetic: in a video game level, a randomly-generated virtual tree might need to frame a house in just the right way. Or they might be functional: the house itself might be randomly-generated, and it needs to be stable when subjected to gravity...so that the player can knock it down later in a tussle with aliens.

To those familiar with probabilistic reasoning, achieving directable randomness might sound suspiciously like a conditional inference problem: we want to sample executions of a random process, subject to a condition or constraint being satisfied. From a Bayesian point of view, running a procedural model is just sampling from some prior distribution $p(\mathbf{x})$, where $\mathbf{x}$ are the random choices made by the procedural model. The constraints can be modeled as a likelihood function $\ell(\mathbf{x})$, where a higher value of ℓ means the output of the model given random choice values $\mathbf{x}$ better satisfies the constraints. The likelihood may often depend on some other data item $\mathbf{y}$, such as a target shape the procedural model should try to match, in which case the likelihood function is $\ell(\mathbf{y}|\mathbf{x})$. Generating constraint-satisfying model outputs then corresponds to sampling from the distribution $p(\mathbf{x}|\mathbf{y}) = \frac{1}{Z}p(\mathbf{x})\ell(\mathbf{y}|\mathbf{x})$, which can be accomplished using one of a variety of Bayesian inference algorithms.

We believe that probabilistic programming languages (PPLs) are particularly well-suited to this Bayesian inference task. PPLs are a universal representation for probability distributions; they can encode any stochastic process [29]. Procedural models are often complicated, featuring both discrete and continuous variables, large blocks of deterministic computation, and complex control flow (including recursion). The universality of probabilistic programs makes them a good choice of representation. PPLs can also be made easy to use, as writing down a probabilistic model in a PPL requires only a moderate amount of programming skill as well as some domain knowledge about the problem to be solved. In our setting, this means that graphics artists and developers do not have to be inference experts in order to use them: they just have to know how to write code.

Unfortunately, the expressiveness and generality of probabilistic programs comes at a cost: inference is typically inefficient. Very simple programs can sometimes be reduced to Bayesian networks or factor graphs, allowing the application of classical inference techniques for probabilistic graphical models [54]. But as we mentioned

above, procedural modeling programs are not simple. They feature both discrete and continuous random variables in complex dependency patterns. And they are often transdimensional, where the number of random variables is itself a random variable—such phenomena can arise in recursive probabilistic programs. Programs with such properties typically do not have graphical model analogues. Thus, to perform inference, we must fall back to treating the program as a black box from which samples can be drawn, and use slowly-converging, approximate Monte Carlo methods such as Markov Chain Monte Carlo. This inefficiency is disappointing, as we would ideally like to embed PPL inference in interactive applications such as games and design tools.

This dissertation focuses on improving PPL inference efficiency for procedural modeling programs. To do this, we peek inside the probabilistic programming black box, identifying and exploiting common properties of procedural modeling programs to make inference run faster or more efficiently. In the work we present, these properties include incrementalizable subcomputations, derivatives, re-orderable subcomputations, and dataflow paths that can be replaced with neural networks. Exploiting such properties often requires techniques from the programming languages literature, such as program transformations.

1.1 Contributions

In this dissertation, we make the following contributions to improving PPL inference efficiency for procedural modeling and design:

- We present an incremental variant of the Metropolis-Hastings (MH) algorithm for probabilistic programs which eliminates redundant computation incurred by previous approaches. The algorithm is comparatively easy to implement, requiring only a few source-to-source program transformations, rather than an entire custom interpreter. We show how our algorithm, called C3, accelerates MH inference for recursive, tree-structured procedural modeling programs. It also offers performance benefits for classical machine learning and Bayesian data analysis programs.

- We apply the Hamiltonian Monte Carlo algorithm to procedural modeling and design programs, demonstrating that it allows efficient exploration of tightly-constrained design spaces where Metropolis-Hastings fails. We evaluate the algorithm on two different families of programs, including programs for assembling 3D structures that can jointly infer the spatial configuration of a structure as well as internal forces which make the structure stable.

- We develop a new variant of the Sequential Monte Carlo algorithm for procedural modeling programs. This new algorithm, called Stochastically-Ordered Sequential Monte Carlo (SOSMC), randomly re-orders exchangeable subcomputations within the program. We show that SOSMC finds higher-likelihood samples with less computation time than either SMC or MH.

- We introduce neurally-guided procedural models: procedural modeling programs whose random choices are controlled by neural networks, rather than the dataflow of the original program. As a precomputation step, we show how to train these networks using high-likelihood samples generated by an inference algorithm such as SMC. Using the trained networks as an importance sampler, SMC can find high-likelihood samples 5-10x faster.

Together, these contributions form the basis of an inference toolkit that brings interactive inference for procedural modeling and design closer to reality. Our implementations for all of these techniques are open-source; links to source code are provided in the respective chapters for each method. This technical work is based primarily on four previous publications [92, 90, 91, 93].

1.2 Dissertation Overview

The rest of this dissertation is organized as follows:

In Chapter 2, we cover necessary background information for our technical contributions. This material includes the history of probabilistic methods for directable procedural content in computer graphics, as well as the basics of probabilistic programming languages and the different techniques used to implement them. To make

things more concrete, we then provide three detailed examples of procedural models expressed as probabilistic programs: furniture layout, color palette design, and constrained 3D trees. We use the challenging characteristics of these programs to motivate the need for more efficient inference.

The next four chapters present our technical contributions in detail. Chapter 3 describes C3, our incremental variant of Metropolis-Hastings. Chapter 4 covers the Hamiltonian Monte Carlo algorithm and our application of it to design space exploration. Chapter 5 details the Stochastically-Ordered Sequential Monte Carlo algorithm and its use in controlling 3D model-generating programs. Chapter 6 introduces the neurally-guided procedural modeling paradigm.

Finally, we conclude the dissertation in Chapter 7, where we summarize our main contributions and discuss the future of probabilistic programming for procedural modeling and design.

Chapter 2

Background

In this chapter, we briefly review the most relevant background material to put our technical contributions in context. We first survey the use of probabilistic methods as applied to directable procedural content creation in computer graphics, painting a picture of the state-of-the-art prior to the application of probabilistic programming to this problem. We then provide a brief primer on probabilistic programming languages, including the common strategies for implementing them, as our contributions build directly on this prior work. Finally, we bring these two areas of research together by presenting three detailed examples of procedural models expressed as probabilistic programs. We point out some properties of these programs that make inference challenging, motivating the need for our contributions to more efficient inference.

2.1 Probabilistic Methods for Directable Procedural Content

While the quest for directable procedural content is an old one, the prevalence of probabilistic techniques for solving the problem is relatively new. In recent years, an increasing number of research projects have relied on probabilistic modeling representations and inference algorithms to solve content generation problems. In this

section, we survey several examples in the domains of object modeling, scene modeling, appearance modeling, and animation.

2.1.1 Object Modeling

Procedural models are most commonly used to generate 3D object geometry. Such procedural object models are often represented using probabilistic grammars, whose hierarchical (and sometimes recursive) branching structure can naturally encode shapes ranging from plants to buildings [86, 74]. In this setting, users may wish to impose some constraints on the generated shapes; several recent projects have used probabilistic inference to provide this control. The Metropolis Procedural Modeling system uses reversible-jump MCMC to infer grammar derivations which are similar to a target image or a target volume. [112]. More recent work focuses on inferring the parameters of procedural trees, using MCMC with a custom likelihood function that measures similarity to a user-provided polygonal tree model [106]. The inferred parameters then allow generation of new trees which resemble the input model. Other work has focused instead on urban building facades, using the same techniques as Metropolis Procedural Modeling to control derivations from a facade-generating split grammar [67]. In this project, the constraint is a target photograph that the generated facade should match as closely as possible.

2.1.2 Scene and Environment Modeling

Virtual objects are not of much use without virtual environments in which to put them: these can range from massive, outdoor terrains to small interior settings such as bedrooms and kitchens. Procedural generation of such environments is becoming increasingly important, as modern games and simulations demand a greater quantity and variety of virtual playing fields. Several projects have explored the generation of indoor furniture layouts by MCMC sampling from a probability distribution that encodes functional and aesthetic constraints [69, 130, 129]. These constraints include criteria such as providing space for a person to walk between furniture items and encouraging visual balance between objects in the room. MCMC has also been used

to control the generation of virtual cities, sampling from a distribution that encodes constraints on how much sunlight buildings receive, how close they are to nearby parks, and other urban planning criteria [114].

2.1.3 Color and Appearance

The examples presented thus far have all focused on creating geometry, but the 'appearance' of that geometry is also important: its color, texture, and material, as well as how it is lit. Procedural methods for these features facilitate automatic generation of richer visual content. The criteria for what constitutes a good or appropriate appearance are often vaguely defined and subjective, so probabilistic models of appearance are often learned from examples. One recent project learns how to color 2D patterns by training a factor graph model on patterns colored by human artists [62]. The factors capture properties that colors should have based on the shape and spatial characteristics of the regions they fill (e.g. background colors are more likely to be very dark or very light). Another system samples outfits for virtual characters by drawing from a large wardrobe of possible garments [131]. Correlations between different types of garments (e.g. dress slacks with suit jackets) are modeled by a Bayesian network trained on labeled photographs of people, and garment colors are sampled from a learned model of appealing color palettes. There has also been work on predicting the material properties of 3D objects (color, shininess, etc.) given only the object's geometry [46]. This system also uses a factor graph model, trained on a large database of objects with materials assigned.

2.1.4 Motion and Animation

Many computer graphics applications require dynamic visual content: virtual objects that move, according to pre-determined animation or in response to user input. Probabilistic control methods have found some applications in procedural animation, as well. In one early example, a rigid-body simulation system uses MCMC to explore possible simulation trajectories until arriving at one that has a desired final state, such as a bouncing die that lands in a particular spot [12]. To make this possible, the

deterministic rigid-body physics simulation is transformed into a stochastic process by randomly varying the bounce direction whenever an object strikes a surface. More recently, another project uses Sequential Monte Carlo for real-time control of a virtual character reacting to external forces [36]. The SMC proposal function advances the character's motion according to a physics simulation, and the likelihood function measures adherence to goals (e.g. not falling down, moving smoothly). Simulating multiple SMC particles in parallel maintains multiple possible motion states, preventing the character from being trapped in a single unstable state when confronted with an unexpected new stimulus. Yet another project uses a variant of Metropolis-Hastings to infer mechanical crankshaft assemblies that, when turned, cause a mechanical toy to produce a desired motion [133].

2.2 Probabilistic Programs

The systems just surveyed demonstrate the potential of probabilistic techniques for controlling procedural graphics content. But they are scattered across many different application domains. They rely on a grab-bag of probabilistic modeling representations, from Bayesian networks to factor graphs to grammars to ad-hoc, problem-specific models. And they employ a host of algorithms, from MCMC to SMC to belief propagation, not all of which are immediately recognizable as such.

Probabilistic programs can unify this effort: since they are a universal representation for probability distributions, they subsume all other types of models used by these systems [29]. They can also be made to support many inference algorithms. They are also an *understandable* representation for probability distributions: computer graphics artists and developers can read and write them without having to be inference experts.

2.2.1 Definition

For the purposes of this thesis, we define a probabilistic programming language (PPL) to be a Turing-complete, deterministic language augmented with the following additional language primitives:

- **Random choices:** Stochastic functions which sample from a probability distribution, e.g. Gaussian or Bernoulli.
- **Conditioning:** Statements which impose some constraint on the values of random choices or other program expressions.
- **Inference:** Operators which compute the conditional distribution over some program value(s) given some constraints.

The following program illustrates these concepts. It is written in WebPPL, a probabilistic programming language based on a subset of Javascript [30]. We use WebPPL for several examples throughout this dissertation:

```
var model = function() {
  var a = flip(0.5);
  var b = flip(0.5);
  var c = flip(0.5);
  condition(a || b);
  return a + b + c;
};
Enumerate(model);
```

In this program, `flip` samples from a Bernoulli distribution, `condition` imposes the constraint that its argument must be true, and `Enumerate` computes the conditional distribution on the output of `model` (i..e `a + b + c`) conditioned on `a || b` being true. The output of this program is the discrete marginal distribution

```
2 : 0.500
1 : 0.333
3 : 0.167
```

indicating that under the constraint, `model` has output 2 half of the time, output 1 one third of time, and output 3 one sixth of the time. Output 0 is has zero probability, as the `condition` statement does not allow for it.

The semantics of inference are defined via rejection sampling: the distribution computed by inference is the distribution on values that results from running the program forward and rejecting return values that do not satisfy the constraints. While a correct specification, this procedure is not a very efficient way to compute conditional distributions, and so most implementations of probabilistic programs use more sophisticated methods under the hood. Additionally, while the `condition` statement imposes a hard constraint, probabilistic programming languages may also provide mechanisms for enforcing soft constraints (in WebPPL, the `factor` statement serves this role). In this case, inference semantics are defined by likelihood weighting [54]. In this thesis, we focus primarily on such soft constraints. Other work has looked in-depth at inference in the presence of hard constraints for procedural content generation [125].

In addition to WebPPL, there are many other existing production and research systems that fit our precise definition of probabilistic programs. These include but are not limited to Church [29], Venture [65], and Anglican [121]. There are other implementations of probabilistic programming which are also Turing-complete but that formulate the semantics of inference differently; these include BLOG [71], Stan [104], and Figaro [81]. Finally, there also exist a number of probabilistic modeling languages which build and perform inference on graphical models, but which are not fully Turing-complete probabilistic programming languages. These include the long-established BUGS [103] and JAGS [82] software packages, as well as Dimple [39] and Factorie [68].

2.2.2 Inference

Probabilistic programming languages demand inference algorithms that work on programs in general, regardless of what phenomena those programs actually model. The following are the most prevalent, general-purpose inference algorithms that have been made to work in the probabilistic programming setting:

Exact inference For programs that make only discrete random choices with finite support, exact inference is possible via enumerating all possible paths through the program and marginalizing out all of the latent random choices. This is in fact the

operation performed by `Enumerate` in the example program above.

Approximate inference When programs include random choices with infinite support, inference engines typically fall back to one of the following approximate algorithms:

- **Rejection sampling / likelihood weighting:** In the simplest cases, inference can proceed via its semantics, by repeatedly drawing forward samples from the program, discarding those that fail hard constraints, and weighting the remainder according to any soft constraints [29].
- **Markov Chain Monte Carlo (MCMC):** When it is extremely unlikely to draw high-likelihood samples via forward sampling, MCMC typically performs better than likelihood weighting. In the PPL setting, MCMC usually takes the form of the Metropolis-Hastings algorithm [70]: the inference engine proposes a change to the value of some random choice(s) in the program, and the new value is propagated to the rest of the program [119, 65].
- **Sequential Monte Carlo (SMC):** a.k.a. particle filtering [21]. A popular choice for time-series models such as hidden Markov models, SMC can be applied to programs in general, since all probabilistic programs involve a sequence of random choices. PPLs implement SMC by running multiple copies of the program (conceptually) in parallel, suspending them at synchronization barriers to perform particle resampling [121, 79].
- **Variational inference:** a family of algorithms which attempts to optimize the free parameters of an easy-to-sample-from 'guide' distribution such that it is similar to the hard-to-sample-from 'target' distribution of interest. Variational inference has recently been applied to probabilistic programs by deriving a guide program using simple transformations of the target program [45, 120, 72].

We focus on approximate algorithms in this thesis—especially MCMC, SMC, and variational inference—since exact inference is not applicable to the complex procedural modeling programs we are interested in developing.

2.2.3 Implementation

PPL inference algorithms require the ability to interact/interfere with program execution: to pause and resume it, to inspect and change the values of random choices, etc. Most existing PPLs have adopted one of two main strategies for compiling/executing probabilistic program code to make this possible:

Interpretation Running a probabilistic program in a custom interpreter allows for arbitrary modifications to program execution at any time. This is the strategy adopted by the original MIT-Church implementation of the Church language [29], Venture [65], and the original implementation of Anglican [121].

Embedding Alternatively, a probabilistic programming language can be embedded in an existing deterministic language via source-to-source compiler transformations. For example, WebPPL's compiler takes probabilistic code and transforms it into deterministic Javascript code. This strategy is also used by Bher Church [119], Quicksand [89], and the revised implementation of Anglican [122]. There are several advantages to this implementation style: less implementation effort, faster execution by leveraging the host language's native runtime, and easy interaction with existing deterministic code in the host language. For these reasons, we focus on the embedded approach in this thesis.

2.3 Procedural Models as Probabilistic Programs

Having surveyed both probabilistic methods in procedural graphics, as well as probabilistic programs, we are now ready to bring the two together and look at some procedural models written as probabilistic programs. In this section, we provide three such examples: one that generates pleasing color palettes, one that generates furniture layouts, and one that generates 3D tree models. Some code is elided for brevity (i.e. utility subroutines).

2.3.1 Example: Furniture Arrangement

Our first example is a program which generates arrangements of 3D tables as might be seen in a virtual coffee shop (Figure 2.1). While it involves 3D objects, this problem is naturally phrased as a 2D layout problem. Thus, our program extracts the radius of the up-aligned bounding cylinder for each object (`utils.getRadius`) and then proceeds to generate 2D layouts by treating the objects as circles.

The main, top-level function for this program is the `genLayout` function. At a high-level, it randomly samples a number of tables; for each table, it chooses a random number of chairs, a random location, and a random orientation. WebPPL programs can also invoke `factor` statements to introduce likelihood terms (i.e. soft contraints). Each invocation of `factor(x)` adds the quantity `x` to the posterior log-probability of the program execution. This program uses `sepFactor` to ensure that furniture objects are spatially separated, `roomInsidenessFactor` to keep all furniture objects inside the bounds of some room shape, and `roomFillFactor` to encourage the furniture objects to fill up the room as much as possible. If we wanted to get more sophisticated, we could also define factors to align furniture along walls or to ensure the existence of a walkable path through the room [129, 69].

Finally, the program invokes WebPPL's `MCMC` routine with the `onlyMAP` option set to `true` to perform approximate *maximum a posteriori* inference. Since the distribution encoded by this program is complex and multimodal, this inference returns some mode of the distribution. The `addDishes` function (not shown) adds a random configuration of plates and cups atop each table as a detail post-process. This form of composition—feeding the result of inference into further probabilistic computation—is a powerful feature enabled by exposing inference as part of the language itself, rather than as a meta-process external to the language. Figure 2.2 shows two example scenes generated by our furniture layout program.

Note that while WebPPL is a purely functional language, it allows imperative assignments to global variables (i.e. the `globalStore`) via a store-passing transform. This feature can simplify programs that require common data to be shared among many subroutine calls (in this case, the `tables` list).

```
var genTables = function(n) {
  if (n > 0) {
    // Sample table position
    var pos = [uniform(-55,55),uniform(-55,55)];
    // Keep the table inside the room
    roomInsidenessFactor(pos, tableRad);
    // Keep tables a minimum distance apart
    map(function(t) {
      separationFactor(pos, t.pos, 3*tableRad);
    }, globalStore.tables);
    // Generate table at this location
    globalStore.tables =
      globalStore.tables.concat([genTable(pos)]);
    // Continue
    genTables(n - 1);
  }
};

var genTable = function(pos) {
  // Sample overall rotation for the table
  var rot = gaussian(0, Math.PI);
  // Sample number of seats
  var nseats = 2 + randomInteger(3);
  // Compute chair positions
  var chairs = mapN(function(i) {
    // Angle / position
    var cang = rot + (i/nseats) * TWOPI;
    var rad = tableRad+chairRad+chairRad*0.5;
    var cpos = add(pos, polar2rect(rad, cang));
    // Rotation / Forward vector
    var cfwd = normalize(sub(pos, cpos));
    var crot = angleBetween([0, 1], cfwd);
    // Keep chair inside the room
    roomInsidenessFactor(cpos, chairRad);
    // Keep chair away from other furniture
    foreach(globalStore.tables, function(t) {
      sepFactor(cpos,t.pos,2*chairRad+tableRad);
      foreach(t.chairs, function(c) {
        sepFactor(cpos,c.pos,3*chairRad);
      });
    });
    // Return chair object
    return { ang: cang, pos: cpos, rot: crot };
  }, nseats);
  // Return table object
  return { pos: pos, rot: rot, chairs: chairs };
}

var tableRad = util.getRadius('table.obj');
var chairRad = util.getRadius('chair.obj');
var room = util.loadRoom('room.txt');
```

```
var gaussFactor = function(x, mu, sigma) {
  var diff = x - mu;
  factor(-(diff*diff) / (sigma*sigma));
};

var sepFactor = function(p1, p2, minSep) {
  gaussFactor(
    Math.max(minSep - dist(p1, p2), 0), 0, 0.5);
};

var roomInsidenessFactor = function(p, r) {
  var d = pointToPolygonDist(p, room.walls);
  var isIn = util.pointInPolygon(p, room.walls);
  var dSigned = (isIn ? -d : d) + r;
  var pad = -3;
  var dist = Math.max(dSigned, pad);
  return gaussFactor(dist, pad, 2);
};

var roomFillFactor = function(tables) {
  // Measure overlap between tables + room
  var tOvers = map(function(table) {
    // Measure overlap between chairs and room
    var cOvers = map(function(chair) {
      return circleRoomOverlap(
        chair.pos, chairRad, room);
    }, table.chairs);
    return circleRoomOverlap(
      table.pos, tableRad, room)
      + reduce(plus, 0, cOvers);
  }, tables);
  var totalOver = reduce(plus, 0, tOvers);
  gaussFactor(
    Math.min(totalOver/room.area, 1), 1, 0.1);
};

var genLayout = function() {
  // Generate tables & chairs
  globalStore.tables = [];
  genTables(poisson(4));
  // Encourage furniture to fill the room
  roomFillFactor(globalStore.tables);
  // Return scene
  return { room: room,
    tables: globalStore.tables
  };
};

// Perform inference for final result
var scene = MCMC(genLayout,
{ samples: 10000, onlyMAP: true }).MAP().val;
addDishes(scene);
```

Figure 2.1: WebPPL program for generating furniture layouts.

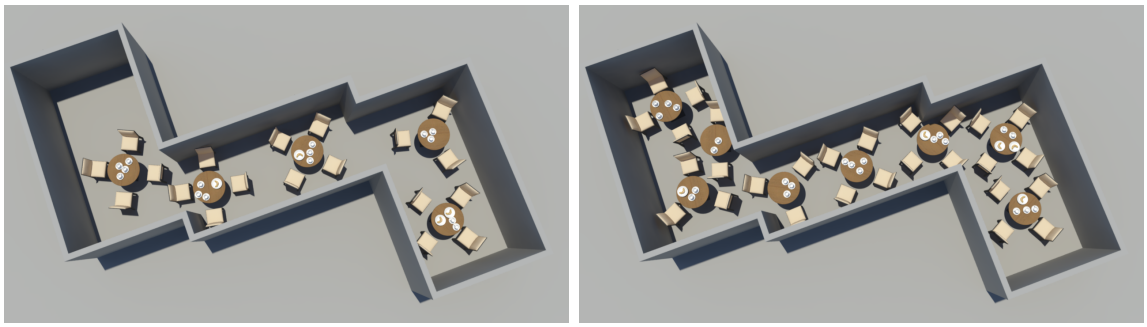


Figure 2.2: Furniture layouts generated by the program in Figure 2.1.

2.3.2 Example: Color Palette Design

For our next example, we consider writing a probabilistic program to generate appealing 5-color palettes. Such a program could be used to automatically create color themes for websites, user interfaces, or interior design projects. 'Appealing' is a nebulous goal, however, and the most recent work on predicting the quality of color palettes does so by learning from a large number of examples [77]. To keep our example simple, we will instead use heuristics to generate a pleasing subset of all possible palettes.

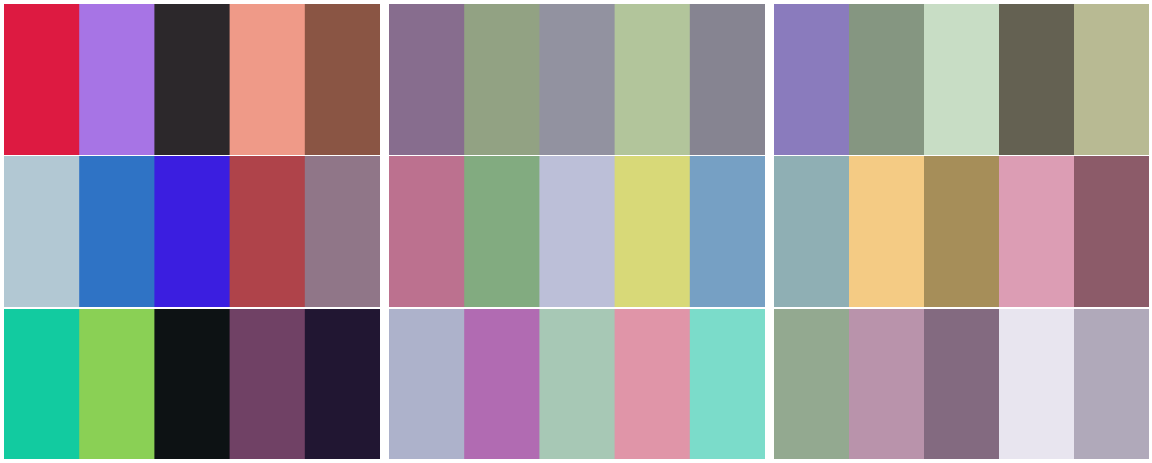
Figure 2.3 shows the code for this program. It begins by sampling five random colors from the HSV color space. If we draw samples from just this prior distribution, we obtain palettes such as those shown in Figure 2.4a. These completely random palettes are not very appealing, so we clearly need to refine our program. We first add some factors to encourage each color to be pastel (see `pastelFactors`) and to ensure that adjacent colors are sufficiently far apart in color space that they don't bleed together (see `adjacencyFactors`). Performing inference on this program now gives results such as those in Figure 2.4b. These colors are legible and more appealing, but they still lack global harmony. To further improve results, we will restrict our palettes to a specific design scheme: one main color (`colors[0]`), two accent colors (`colors[1]` and `colors[3]`), and two shades of those accent colors (`colors[2]` and `colors[4]`). To increase overall color harmony, we add factors to encourage the main and accent colors to be 'split-complementary': that is, forming an isosceles triangle when arranged on a color wheel (see `complementarityFactors`). We then add factors to encourage `colors[2]` and `colors[4]` to be the same shade of `colors[1]` and `colors[3]`, respectively (see `shadeFactors`). Inference on this complete program now produces samples like those in Figure 2.4c, which appear more balanced and appealing.

2.3.3 Example: 3D Tree Modeling

For our final example, we show a program that generates 3D tree skeletons. Trees are one of the most popular applications of procedural modeling, and outdoor environments in games, movies, and simulations often feature forest-sized swaths of

```
var pastelFactors = function(colors) {
  // High value and low saturation
  map(function(c) {
    gaussFactor(c[2], 0.75, 0.06);
    gaussFactor(c[1], 0.25, 0.06);
  }, colors);
};

var adjacencyFactors = function(colors) {
  mapIndexed(function(i, c) {
    if (i < colors.length - 1) {
      var c2 = colors[i + 1];
      gaussFactor(dist(c, c2), 0.5, 0.06);
    }
  }, colors);
};

var cos60 = Math.cos(deg2rad(60.0));
var cos150 = Math.cos(deg2rad(150.0));
var complementarityFactors = function(hv) {
  gaussFactor(dot(hv[0], hv[1]), cos150, 0.01);
  gaussFactor(dot(hv[1], hv[3]), cos60, 0.01);
  gaussFactor(dot(hv[3], hv[0]), cos150, 0.01);
};

var shadeFactors = function(c, hv) {
  // Same hue
  gaussFactor(dot(hv[1], hv[2]), 1.0, 0.01);
  gaussFactor(dot(hv[3], hv[4]), 1.0, 0.01);
  // Same saturation
  gaussFactor(c[1][1] - c[2][1], 0.0, 0.01);
  gaussFactor(c[3][1] - c[4][1], 0.0, 0.01);
  // Two shades have same lightness difference
  var vdiff12 = c[1][2] - c[2][2];
  var vdiff34 = c[3][2] - c[4][2];
  gaussFactor(vdiff12 - vdiff34, 0.0, 0.01);
};

var genPalette = function() {
  // Sample random colors
  // HSV, components in range [0, 1]
  var colors = repeat(5, function() {
    return [
      uniform(0.0, 1.0),   // H
      uniform(0.0, 1.0),   // S
      uniform(0.0, 1.0)    // V
    ];
  });

  // Encourage 'pastel' colors
  pastelFactors(colors);

  // Encourage adjacent color difference
  adjacencyFactors(colors);

  var hueVectors = map(function(c) {
    return polar2rect(1, 2 * Math.PI * c[0]);
  }, colors);

  // Split complementarity
  complementarityFactors(hueVectors);

  // Shades
  shadeFactors(colors, hueVectors);

  return colors;
};

// Perform inference
MCMC(genPalette, {
  samples: 20000,
  onlyMAP: true
}).MAP().val;
```

Figure 2.3: WebPPL program for generating color palettes.



(a) (b) (c)

Figure 2.4: Palettes generated by the program in Figure 2.3 with *(a)* just the prior distribution, *(b)* pastel and adjacency factors added, and *(c)* all factors added.

procedurally-generated vegetation [43].

Figure 2.5 shows the code for this program; `genTree` is the top-level function. It calls the recursive `branch` function, each invocation of which generates a single piece of tree branch geometry. These geometries accumulate in the `globalStore.geometry` list. `branch` itself is structured much like an L-system [85]: it maintains a current coordinate frame (`curr`) describing the position, orientation, and size of the last branch generated. Simple random processes determine whether to continue the current branch, whether to generate an offshoot branch, as well as the continuous parameters of these new branches. Figure 2.6a shows an example output from this process.

This program also controls the shape of the generated trees through factors that encourage similarity to a target volume. The program rasterizes the generated geometry in `globalStore.geometry` onto a 3D voxel grid and performs a cell-by-cell comparison between the resulting grid and a target grid. The result of this comparison feeds into a `factor` statement that penalizes dissimilarity. Figure 2.6c show a sample from this posterior distribution, given the volume target in Figure 2.6b.

2.3.4 Inference Challenges

In presenting these three examples, we focused on how to express procedural models as probabilistic programs. In the process, we have glossed over some of the ugly details about how inference actual performs on programs such as these.

As show in the example code, we used MCMC inference for all three examples. In particular, WebPPL uses a popular variant of Metropolis-Hastings for probabilistic programs called "lightweight MH" [119]. This algorithm, while elegant and simple to implement, unfortunately performs a significant amount of redundant computation, meaning that inference on our programs takes longer to run than is strictly required. In Chapter 3, we propose a new extension to lightweight MH that eliminates much of this redundant computation. It is especially helpful for recursive programs such as the tree program in Figure 2.5.

Faster MH is not always enough. For each column in Figure 2.4, we generated the three color palettes using three separate runs of MH. Ideally, we would generate

```
var branch = function(r0, curr, i, d, prev) {
  // Stop generating if branches get too small
  if (curr.radius / r0 >= 0.1) {
    var uprot = gaussian(0, Math.PI / 12);
    var leftrot = gaussian(0, Math.PI / 12);
    var len = uniform(3, 5) * curr.radius;
    var endradius = uniform(0.7, 0.9) *
      curr.radius;

    // Tree segments represented by two
    //   connected conic sections
    var next = utils.advanceFrame(
      curr, uprot, leftrot, len, endradius);
    var split = utils.findSplitFrame(
      curr, next);
    var geom = utils.treeSegment(
      prev, curr, split, next);
    globalStore.geometry =
      globalStore.geometry.concat([geom]);

    // Recursively branch?
    if (flip(0.5)) {
      // Branches more likely on upward-facing
      //   parts of parent branch
      var upnessDistrib =
        utils.estimateUpness(split, next);
      var theta = gaussian(
        upnessDistrib[0], upnessDistrib[1]);
      var branchradius = uniform(0.9, 1) *
        endradius;
      // Branches spawn in middle of parent
      var t = 0.5;
      var b = utils.branchFrame(
        split, next, t, theta, branchradius);
      branch(r0, b.frame, 0, d + 1, b.prev);
    }

    // Keep generating same branch?
    if (flip(Math.exp(-0.1*i)))
      branch(r0, next, i + 1, d, null);
  }
};
```

```
var volTgt = io.loadVolumeTarget(
  'targets/treeProxy_long_2.obj');

var genTree = function() {

  globalStore.geometry = [];

  var start = {
    center: Vector3(0, 0, 0),
    forward: Vector3(0, 1, 0),
    up: Vector3(0, 0, -1),
    radius: uniform(1.5, 2)
  };

  branch(start.radius, start, 0, 0, null);

  var geom = utils.geomerge(globalStore.geometry);

  // Zero probability if self-intersects
  if (geom.selfIntersects())
    factor(-Infinity);

  // Encourage volumetric similarity
  var resolution =
    volTgt.targetGrid.dims;
  var grid = utils.VoxelGrid(resolution);
  geom.voxelize(grid, volTgt.bounds,
    resolution, true);
  var percentSame =
    volTgt.targetGrid.percentCellsEqual(grid);
  gaussFactor(
    percentSame,
    1,
    volTgt.percentSameSigma)

  return geom;
};

// Perform inference
MCMC(genTree,
  {samples: 1000, onlyMAP: true}).MAP().val;
```

Figure 2.5: WebPPL program for generating 3D tree skeletons.

(a) (b) (c)

Figure 2.6: 3D tree skeletons generated by the program in Figure 2.5. *(a)* A sample from the prior distribution. *(c)* A sample from the posterior distribution, conditioned on being volumetrically similar to the shape in *(b)*.

multiple such distinct palettes using a *single* run of the algorithm, but this does not work: MH quickly gets stuck in one mode of the posterior distribution. Why does this occur? `complementarityFactor` introduces a tight constraint between multiple palette colors, and MH cannot make constraint-respecting coordinated changes to all of them at once. In Chapter 4, we show how to get around design problems like this one using Hamiltonian Monte Carlo.

Tight constraints make it difficult to explore the space of program outputs, but sometimes it can be challenging to find even *one* high-probability output. The recursive, branching nature of the tree program in Figure 2.5 results in an exponentially-large space of possible tree structures; long sequences of random choices must be carefully coordinated to find a structure which matches the target volume. MCMC, perhaps unsurprisingly, cannot perform this task reliably, and so requires many iterations to converge to a matching result. In Chapter 5, we show how to more quickly and reliably sample from programs like this one using a new variant of Sequential Monte Carlo.

Finally, whether we use MCMC or SMC, both are random search procedures. In Chapter 6, we explore one way of making this search less random: training neural networks to help the program quickly seek out high-probability execution traces.

Chapter 3

Eliminating Redundant Computation in MCMC

As mentioned in Section 2.2, there are many possible implementations of PPL inference, but one popular choice is the 'Lightweight MH' framework [119]. Lightweight MH uses a source-to-source transformation to turn a probabilistic program into a deterministic one, where random choices are uniquely identified by their structural position in the program execution trace. Random choice values are then stored in a database indexed by these structural 'addresses.' To perform a Metropolis-Hastings proposal, Lightweight MH changes the value of a random choice and re-executes the program, looking up the values of other random choices in the database to reuse them when possible. Lightweight MH is simple to implement and allows PPLs to be built atop existing deterministic languages. Users can thus leverage existing libraries and fast compilers/runtimes for these 'host' languages. For example, Stochastic Matlab can access Matlab's rich matrix and image manipulation routines [119], WebPPL runs on Google's highly-optimized V8 Javascript engine [30], and Quicksand's host language compiles to fast machine code using LLVM [89].

Unfortunately, Lightweight MH is also inefficient: when an MH proposal changes a random choice, the entire program re-executes to propagate this change. This is rarely necessary: for many models, most proposals affect only a small subset of the program

execution trace. To update the trace, re-execution is needed only where values can change. Under Lightweight MH, random choice values are preserved and reused when possible, limiting the effect of a proposal to a subset of the changed variable's Markov blanket (sometimes a much smaller subset, due to context-specific independence [7]). Custom PPL interpreters can leverage this property to incrementalize proposal re-execution [65], but implementing such interpreters is complicated, and using them makes it difficult or impossible to leverage libraries and fast runtimes for existing deterministic languages.

In this chapter, we present a new implementation technique for MH proposals on probabilistic programs that gives the best of both worlds: incrementalized proposal execution using a lightweight, source-to-source transformation framework. It is especially successful at accelerating recursive probabilistic programs with many local latent variables. Our method, C3, is based on two core ideas:

1. *Continuations*: Converting the program into continuation-passing style to allow program re-execution to begin anywhere.

2. *Callsite caching*: Caching function calls to avoid re-execution when function inputs or outputs have not changed.

We first describe how to implement C3 in any functional PPL with first-class functions. Our implementation is integrated into the open-source WebPPL probabilistic programming language [30]; it requires only small changes to how WebPPL programs are normally written. We then compare C3 to Lightweight MH, showing that it gives orders of magnitude speedups on common models such as HMMs, topic models, Gaussian mixtures, and hierarchical linear regression. In some cases, C3 reduces runtimes from linear in model size to constant. We also demonstrate that C3 is nearly an order of magnitude faster on a complex inverse procedural modeling example.

3.1 Approach

To illustrate our approach, we use a simple example: a binary state Hidden Markov Model program written in WebPPL (Figure 3.1 Left). This program recursively

```
1  // Hidden Markov Model
2  var hmm = function(n, obs) {
3    if (n === 0)
4      return true;
5    else {
6      var prev = hmm(n-1, obs);
7      var state = transition(prev);
8      observation(state, obs[n]);
9      query.add(n, state);
10     return state;
11   }
12 };
13 hmm(100, observed_data);
14 return query;
```

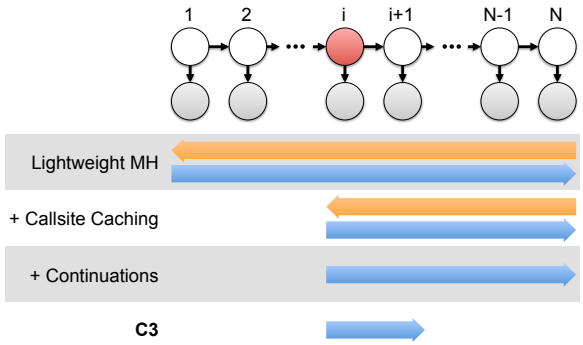


Figure 3.1: *(Left)* A simple HMM program in the WebPPL language; the highlighted lines involving `query` are the only modifications necessary to use our method with this program. *(Right)* Illustrating the re-execution behavior of different MH implementations in response to a proposal to the random choice c_i shaded in red. Lightweight MH re-executes the entire `hmm` program, invoking (orange bar) and then unwinding (blue bar) the full chain of recursive calls. Callsite caching allows re-execution to skip all recursive calls under `hmm(i-1, obs)`. With continuations, re-execution only has to unwind from the continuation of choice c_i. Combining callsite caching and continuations allows re-execution to terminate upon returning from `hmm(i+1, obs)`, since its return value does not change.

samples latent states (inside the `transition` function), conditioning on the observations in the `obs` list (inside the `observation` function). When invoked, `hmm(N, obs)` generates a linear chain of latent and observed random variables (Figure 3.1 Right). The values of the latent state variables are stored in the special `query` table; we will show later how this small modification allows our method to be used with this program.

Consider how Lightweight MH performs a proposal on this program. It first runs the program once to initialize the database of random choices. It then selects a choice c_i uniformly at random from this database (the red circle in Figure 3.1 Right) and changes its value. This change necessitates a constant-time update to the score of c_{i+1}. However, Lightweight MH re-executes the *entire* program, invoking a chain of recursive calls to `hmm` (the orange bar in Figure 3.1 Right) and then unwinding those calls (the blue bar). This process requires $2N$ such call visits for an HMM with N states.

One strategy for speeding up re-execution is to cache function calls and reuse their results if they are invoked again with unchanged inputs. We call this scheme, which is a generalization of Lightweight MH's random choice reuse policy, *callsite caching*. With this strategy, the recursive re-execution of `hmm` must still traverse all ancestors of choice c_i but can stop at `hmm(i, obs)`: it can reuse the result of `hmm(i-1, obs)`, since the inputs have not changed. As shown in Figure 3.1 Right, using callsite caching can result in less re-execution, but it still requires $\sim 2N$ `hmm` call visits on average.

Now suppose we instead convert the program into continuation passing style. CPS re-organizes a program to make all data and control flow explicit—instead of returning, functions invoke a 'continuation' function which represents the remaining computation to be performed [2]. For our HMM example, by storing the continuation at c_i, computation can resume from the point where this random choice is made, which corresponds to unwinding the stack from `hmm(i, obs)` up to `hmm(N, obs)`. Looking at the 'Continuations' row of Figure 3.1, this is a significant improvement over Lightweight MH and is also better than callsite caching. However, it still requires $\sim N$ call visits.

Our main insight is that we can achieve the desired runtime by combining callsite caching with continuations—we call the resulting system **C3**. With C3, re-execution can not only jump directly to choice c_i by invoking its continuation, but it can actually terminate almost immediately: the cache also contains the return values of all function calls, and since the return value of `hmm(i+1, obs)` has not changed, all subsequent computation will not change either. C3 unwinds only two recursive `hmm` calls, giving the desired constant-time update. Despite this early termination, the values of all hidden states are still available in the special `query` table (see Section 3.3.3). Thus C3 is more than the sum of its parts: by combining caching with CPS, it enables incrementalization benefits that neither component can deliver independently.

In the sections that follow, we describe how to implement C3 in a functional PPL. Specifically, we describe how to transform the program source at compile-time (Section 3.2) to make requisite data available to the runtime caching mechanism (Section 3.3).

3.2 Compile-time Source Transformations

```
// Initial HMM code
var hmm = function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = hmm(n-1, obs);
    var state = transition(prev);
    observation(state, obs[n]);
    return state;
  }
};
```

```
// After caching transform
var hmm = function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = cache(hmm, n-1, obs);
    var state = cache(transition,
prev);
    cache(observation, state, obs[n]);
    return state;
  }
};
```

```
// After function tagging transform
var hmm = tag(function(n, obs) {
  if (n === 0)
    return true;
  else {
    var prev = cache(hmm, n-1, obs);
    var state = cache(transition, prev);
    cache(observation, state, obs[n]);
    return state;
  }
}, '1', [hmm, transition,
observation]);
```

Figure 3.2: Source code transformations used by C3. *(Left)* Original HMM code. *(Middle)* Code after applying the caching transform, wrapping all callsites with the `cache` intrinsic. *(Right)* Code after applying the function tagging transform, where all functions are annotated with a lexically-unique ID and the values of their free variables.

Lightweight MH transforms the source code of probabilistic programs to compute random choice addresses; the transformed code can then be executed on existing runtimes for the host deterministic language. C3 fits into this framework by adding three additional source transformations: caching, function tagging, and a standard continuation passing style transform for functional languages.

Caching This transform wraps every function callsite with a call to an intrinsic `cache` function (Figure 3.2 Middle). This function performs run-time callsite cache lookups, as described in Section 3.3. We initially left this step to the user, requiring them to hand-annotate calls that should be cached. However, we found that with a simple automatic cache adaptation scheme, the automatic transformation can achieve performance close to that of the optimal hand-annotation without additional user overhead (see Section 3.3.4)

Function tagging This transform analyzes the body of each function and tags the function with both a lexically-unique ID as well as the values of its free variables (Figure 3.2 Right). In Section 3.3, we describe how C3 uses this information to decide whether a function call must be re-executed.

The final source transformation pipeline is: caching → function tagging → address computation → CPS. Standard compiler optimizations such as inlining, constant folding, and common subexpression elimination can then be applied. In fact, the host language compiler often already performs such optimizations, which is an additional benefit of the lightweight transformational approach.

3.3 Runtime Caching Implementation

When performing an MH proposal, callsite caching aims to avoid re-executing functions and to enable early termination from them as often as possible. In this section, we describe how C3 efficiently implements both of these types of computational 'short-circuiting' for probabilistic functional programs. Figure 3.3 provides high-level code for the main subroutines which govern the caching system.

3.3.1 Cache Representation

We first require an efficient cache structure to minimize overhead introduced by performing a cache access on every function call. C3 uses a tree-structured cache: it stores one node for each function call, where a node's children correspond to the function's callees. Random choices are stored as leaf nodes. Thus, the size of the cache is proportional to the runtime of one complete execution of the program.

C3 also maintains a stack of nodes which tracks the program's call stack (`nodeStack` in Figure 3.3). During cache lookups, the desired node, if it exists, must be a child of the node on the top of this stack. Exploiting this property accelerates lookups, which would otherwise proceed from the cache root. Altogether, this structure provides expected constant time lookups, additions, and deletions. In addition, by storing a node's children in execution order, C3 can efficiently determine when child nodes have become 'stale' (i.e. unreachable) due to control flow changes and should be removed. A child node is marked unreachable when its parent begins or resumes execution (`execute` line 8; `propagate` line 22) and marked reachable when it is executed (`execute` line 2). Any children left marked unreachable when the parent exits are removed

```
1  // Arguments added by compiler:
2  // a: current address
3  // k: current continuation
4  function cache(a, k, fn, args) {
5    // Global function call stack
6    var currNode = nodeStack.top();
7    var node = find(a, currNode.children);
8    if (node === null) {
9      node = FunctionNode(a);
10     // Insert maintains execution order
11     insert(node, currNode.children,
12            currNode.nextChildIndex);
13   }
14   execute(node, k, fn, args);
15 }
16
17 // rc: a random choice node
18 function propagate(rc) {
19   // Restore call stack up to rc.parent
20   restore(nodeStack, rc.parent);
21   // Changes to rc may make siblings unreachable
22   markUnreachable(rc.parent.children, rc.index);
23   // Continue executing
24   rc.parent.nextChildIndex = rc.index + 1;
25   rc.k(rc.val);
26 }
```

```
1  function execute(node, k, fn, args) {
2    node.reachable = true; node.k = k;
3    node.index = node.parent.nextChildIndex;
4    // Check for input changes
5    if (!fnEquiv(node.fn, fn) || !equal(node.args, args)) {
6      this.fn = fn; this.args = args;
7      // Mark all children as initially unreachable
8      markUnreachable(this.children, 0);
9      // Call fn with special continuation
10     node.nextChildIndex = 0;
11     nodeStack.push(node);
12     node.entered = true;
13     fn(args, function(retval) {
14       node = nodeStack.pop();
15       // Remove unreachable children
16       removeUnreachables(node.children);
17       // Terminate early on proposals where
18       //    retval does not change
19       var rveq = equal(retval, this.retval);
20       if (!node.entered && rveq) kexit();
21       else {
22         node.entered = false;
23         // retval change may make siblings unreachable
24         if (!rveq)
25           markUnreachable(node.parent.children,
26                           node.index);
27         // Continue executing
28         node.retval = retval;
29         node.parent.nextChildIndex++;
30         k(node.retval);
31       }
32     });
33   } else {
34     node.parent.nextChildIndex++;
35     k(node.retval);
36   }
37 }
```

Figure 3.3: The main subroutines governing C3's callsite cache. Function calls are wrapped with `cache`, which retrieves (or creates) a cache node for a given address `a`. It calls `execute`, which examines the function call's inputs for changes and runs the call if needed. Finally, MH proposals use `propagate` to resume re-execution of the program from a particular random choice node which has been changed.

from the cache (`execute` line 16).

3.3.2 Short-Circuit On Function Entry

As described in Section 3.2, every function call is wrapped in a call to `cache`, which retrieves (or creates) a cache node for the current address. C3 then evaluates whether the node's associated function call must be re-evaluated or if its previous return value can be re-used (the `execute` function). Reuse is possible when the following two criteria are satisfied:

1. The function's arguments are equivalent to those from the previous execution.
2. The *function itself* is equivalent to that from the previous execution.

The first criterion can be verified with conservative equality testing; C3 uses shallow value equality testing, though deeper equality tests could result in more reuse for structured argument types. Deep equality testing is more expensive, though this can be mitigated using data structure techniques such as hash consing [32] or compiler optimizations such as global value numbering [95].

The second criterion is necessary because C3 operates on languages with first-class functions, so the identity of the caller at a given callsite is a runtime variable. Checking whether the two functions are exactly equal (i.e. refer to the same closure) is too conservative, however. Instead, C3 leverages the information provided by the function tagging transform from Section 3.2: two functions are equivalent if they have the same lexical ID (i.e. came from the same source location) and if the values of their free variables are equal. C3 applies this check recursively to any function-valued free variables, and it also memoizes the result, as program execution traces often feature many applications of the same function. This scheme is especially critical to obtain reuse in programs that feature anonymous functions, as those manifest as different closures for each program execution.

3.3.3 Short-Circuit On Function Exit

When C3 re-executes the program after changing a random choice (using the `propagate` function), control may eventually return to a function call whose return value has not changed. In this case, since all subsequent computation will have the same result, C3 can terminate execution early by invoking the exit continuation `kexit`. During function exit, C3's `execute` function detects if control is returning from a proposal by checking if the call is exiting without having first been entered (line 20). This condition signals that the current re-execution originated at some descendant of the exiting call, i.e. a random choice node.

Early termination is complicated by inference queries whose size depends on model size: for example, the sequence of latent states in an HMM. In lightweight PPL

implementations, inference typically computes the marginal distribution on program return values. Thus, a naïve HMM implementation would construct and return a list of latent states. However, this implementation makes early termination impossible, as the list must be recursively reconstructed after a change to any of its elements.

For these scenarios, C3 offers a solution in the form of a global `query` table to which the program can write values of interest. Critically, `query` has a *write-only* interface: since the program cannot read from `query`, a write to it cannot introduce side-effects in subsequent computation, and thus the semantics of early termination are preserved. Programs that use `query` can then simply return it to infer the marginal distribution over its contents.

3.3.4 Automatic Cache Adaptation

It is not always optimal to cache *every* callsite: caching introduces overhead, and some function calls almost always change on each invocation. C3 detects such callsites and stops caching them in a heuristic process we call *adaptive caching*. A callsite is un-cached if, after at least N proposals, execution has reached it M times without resulting in either short-circuit-on-entry or short-circuit-on-exit. We use $N = 10, M = 50$ for the results presented in this chapter. With these settings, we have observed modest reductions in both cache size (20–65%) and running time (10–45%). These results are also close to those given by the optimal hand-annotation of `cache` statements. For example, on the LDA example presented in Section 3.4, the automatically-adapted program has runtime within 7% of the optimally hand-annotated program. A small, constant running time overhead remains for un-cached callsites, as calling them still triggers a table lookup to determine their caching status. Future work could explore efficiently re-compiling the program to remove `cache` calls around such callsites.

3.3.5 Optimizations

C3 takes care to ensure that the amount of work it performs in response to a proposal is only proportional to the amount of the program execution trace affected by that proposal. First, it maintains references to all random choices in a hash table, which

provides expected constant time additions, deletions, and random element lookups. This table allows C3 to perform uniform random proposal choice in constant time, rather than the linear time cost of scanning through the entire cache.

Second, proposals may be rejected, which necessitates copying the cache in case its prior state must be restored on rejection. C3 avoids copying the entire cache using a copy-on-write scheme with similar principles to transactional memory [38]: modifications to a cache node's properties are staged and only committed if the proposal is accepted. Thus, C3 only copies as much of the cache as is actually visited during proposal re-execution.

Finally, continuations which never return may overflow the call stack for long-running programs. Our implementation avoids this problem via a standard *trampolining* optimization: instead of directly invoking its continuation, a CPS'ed function returns a thunk (i.e. a nullary function) which encapsulates the continuation. The program repeatedly calls the series of returned thunks in a loop, thus executing the program with only one function call on the stack at any time.

3.4 Experimental Results

We now investigate the runtime performance characteristics of C3. We compare C3 to Lightweight MH, as well as to systems that use only callsite caching and only continuations. This allows us to investigate the incremental benefit provided by each of C3's components. Our implementation of C3 itself is available as part of the WebPPL probabilistic programming language [30]. All timing data was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.10.2.

We first evaluate these systems on two standard generative models: a discrete-time Hidden Markov Model and a Latent Dirichlet Allocation model. We use synthetic data, since we are interested purely in the computational efficiency of different implementations of the same statistical inference algorithm. The HMM program uses 10 discrete latent states and 10 discrete observable states and returns the sequence of latent states. We condition it on a random sequence of observations, of increasing length from 10 to 100, and run each system for 10000 MH iterations, collecting a

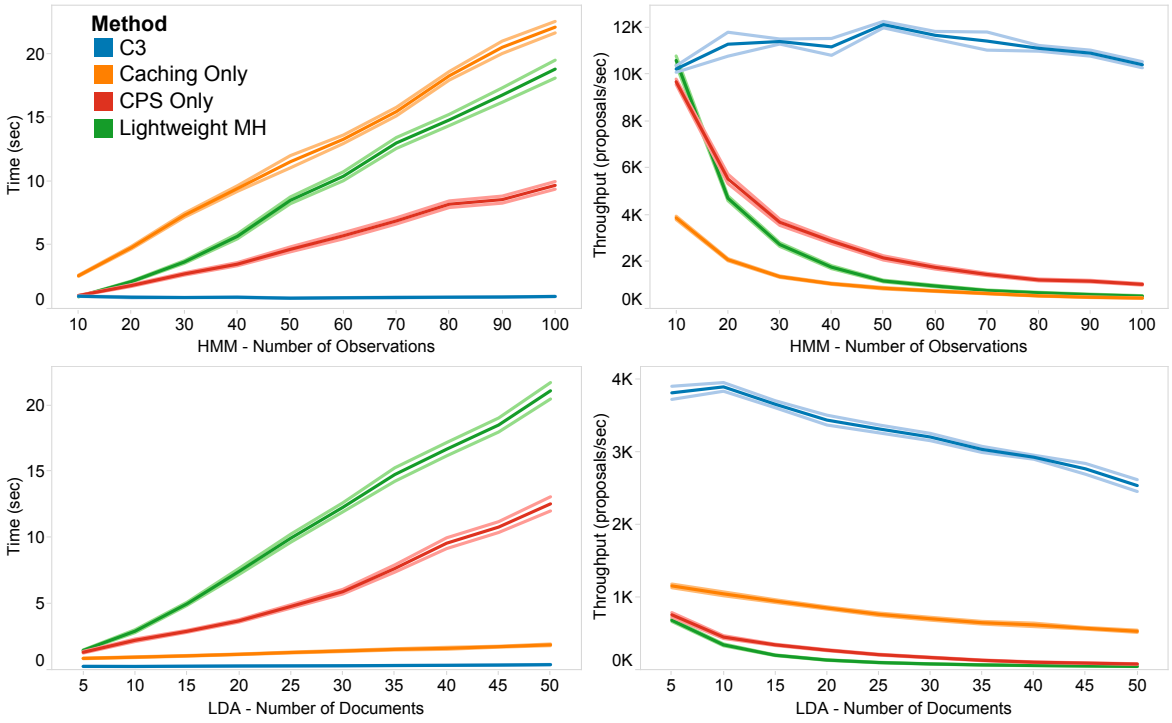


Figure 3.4: Comparing the performance of C3 with other MH implementations. *(Top)* Performing 10000 MH iterations on an HMM program. *(Bottom)* Performing 1000 MH iterations on an LDA program. *(Left)* Wall clock time elapsed, in seconds. *(Right)* Sampling throughput, in proposals per second. 95% confidence bounds are shown in a lighter shade. Only C3 exhibits constant asymptotic complexity for the HMM; other implementations take linear time, exhibiting decreasing throughput.

sample every 10 iterations. The LDA program uses 10 topics, a vocabulary of 100 words, and 20 words per document. It returns the distribution over words for each topic. We condition it on a set of random documents, increasing in size from 5 to 50, and run each system for 1000 MH iterations.

Figure 3.4 shows the results of this experiment; all quantities are averaged over 20 runs. We show wall clock time in seconds (left) and throughput in proposals per second (right). For the HMM, C3's runtime is constant regardless of model size, whereas *Lightweight MH* and *CPS Only* exhibit the expected linear runtime (approximately $2N$ and N, respectively). As discussed in Section 3.1, *Caching Only* has the same complexity as *Lightweight MH* but is a constant factor slower due to

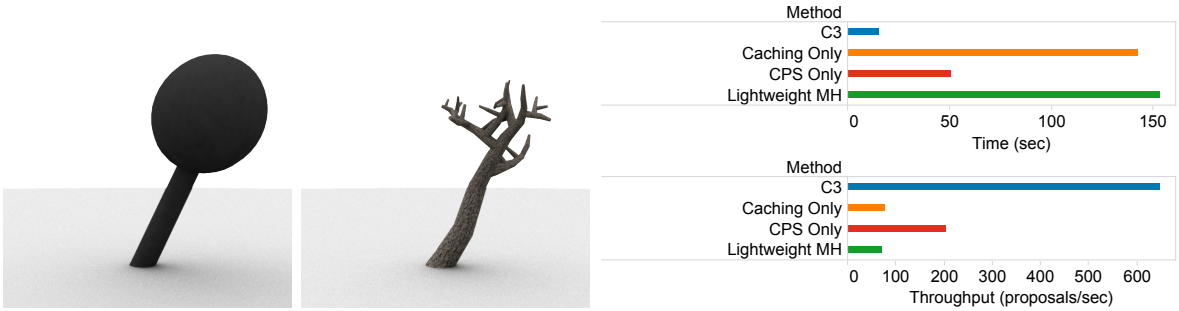


Figure 3.5: Comparing C3 and Lightweight MH on an inverse procedural modeling program. *(Left)* Desired tree shape. *(Middle)* Example output from inference over a tree program given the desired shape. *(Right)* Performance characteristics of different MH implementations. C3 delivers nearly an order of magnitude speedup.

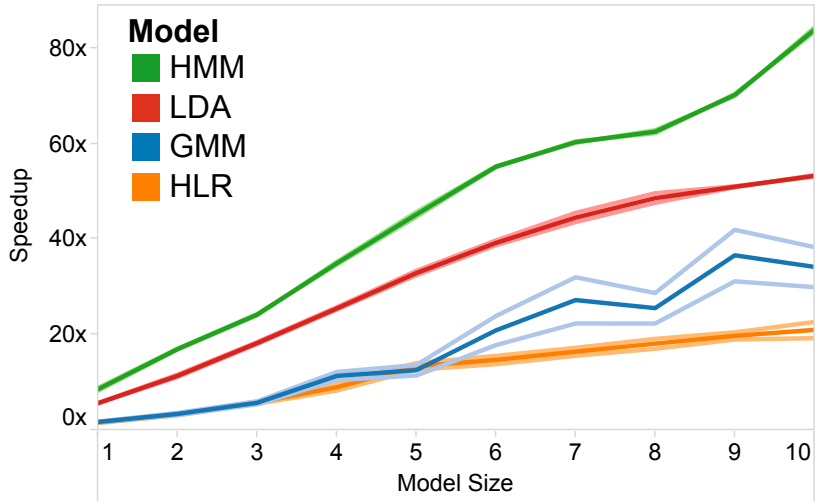
caching overhead. For the LDA model, *Lightweight MH* and *CPS Only* all exhibit asymptotic complexity comparable with their performance on the HMM. However, *Caching Only* performs significantly better. The LDA program is structured with nested loops; caching allows re-execution to skip entire inner loops for many proposals. *Caching Only* must still re-execute all ancestors of a changed random choice, though, so it is slower than C3, which jumps directly to the change point. C3 does not achieve exactly constant runtime for LDA because a small percentage of its proposals affect hierarchical variables, requiring more re-execution. This is a characteristic of hierarchical models in general; in this specific case, conjugacy could be leveraged to integrate out higher-level variables.

We also evaluate these systems on an inverse procedural modeling program. Procedural models are programs that generate random 3D models from the same family. *Inverse* procedural modeling infers executions of such a program that resemble a target output shape [112]. We use a simple grammar-like program for tree skeletons presented in prior work, conditioning its output to be volumetrically similar to a target shape [91]. We run each system for 2000 MH iterations.

Figure 3.5 shows the results of this experiment. C3 achieves the best performance, delivering nearly an order of magnitude speedup over Lightweight MH. Using caching only does not help in this example, since re-executing the program from its beginning reconstructs all of the recursive procedural modeling function's structured inputs,

whose equality is not captured by our cache's shallow equality tests.

Finally, the figure below shows the results of a wider evaluation: for four models, we plot the speedup obtained by C3 over Lightweight MH (in relative throughput) as model size increases. The four models are: the HMM and LDA models from Figure 3.4, a one-dimensional finite Gaussian mixture model (GMM), and a hierarchical linear regression model (HLR) [126]. The 1-10 normalized Model Size parameter maps to a natural scale parameter for each of the four models; details are available in Appendix A. While C3 offers only small benefits over Lightweight MH for small models, it achieves dramatic speedups of 20-100x for large models.



3.5 Related Work

The ideas behind C3 have connections to other areas of active research. First, incrementalizing MCMC proposals for PPLs falls under the umbrella of *incremental computation* [87]. Much of the active work in this field seeks to build general-purpose languages and compilers to incrementalize any program [11]. However, there are also systems such as ours which seek simpler solutions to domain-specific incrementalization problems. In particular, C3's callsite caching mechanism was inspired in part by recent work in computer graphics on hierarchical render caches [123].[1]

[1]An incomplete, undocumented version of C3's callsite caching mechanism also appears in the original MIT-Church implementation of the Church probabilistic programming language [29].

C3 bears similarity to the CPS-based self-adjusting computation system developed by Ley-Wild and colleagues [60]. Both this system and C3 use continuations to approximate dynamic data dependencies, and both use some form of function caching to avoid re-executing unchanged computations. Their system aims for generality, using a compiler infrastructure that supports arbitrary changeable data and computation reuse. The restricted needs of our application—MH proposal computation—allow C3 to use a simpler strategy: only random choices are changeable, and computation is only reused from the previously-accepted program execution. This approach is consistent with our goal of keeping the system lightweight. It also has efficiency benefits: since Ley-Wild's system allows arbitrary changes to data, its change propagation mechanism must examine all of the previous execution's reads and writes to changeable data to ensure that they are consistent with the data's current value. In contrast, C3 knows that a proposal makes exactly one change (i.e. to a random choice value), so it can start change propagation from the continuation at that point, as well as terminate change propagation as soon as any function's return value is unchanged. This ability, along with efficient random choice lookup and cache copy-on-write, enables asymptotically constant-time proposals when the model's dependence structure supports them.

The Venture PPL features an algorithm to incrementally update a probabilistic execution trace in response to a random choice change [65]. Implemented as part of a custom interpreter, this method walks the trace starting from the changed node, identifying nodes which must be updated or removed, and determining when re-evaluation can stop. C3 performs a similar computation but uses continuations to traverse the execution trace rather than maintaining a complete interpreter state.

The Shred system also incrementalizes MH updates for PPLs [126]. Shred traces a program to remove its control flow and then uses data-flow analysis to produce incremental update procedures for each random choice. This process produces very fast proposal code, but it requires significant implementation cost, and its re-compilation overhead grows very large for programs with high control-flow variability, such as PCFGs. C3's caching scheme is a dynamic analog to Shred's static slicing which does not have compilation overhead but may not be as fast for models with fixed control

flow.

The Swift compiler for the BLOG language is another recent system supporting incrementalized MCMC updates [61]. Unlike the above systems, BLOG/Swift uses a *possible-world semantics* for probabilistic programs, representing program state as a graphical model whose structure changes over time. Swift tracks the Markov Blanket of this model, computing incremental updates to it as model structure changes, allowing it to make efficient MCMC proposals. C3 does not explicitly compute Markov blankets, but its short-circuiting facilities limit re-execution to the subset of a changed variable's Markov blanket that is affected by the change.

3.6 Chapter Summary

This chapter presented C3, a lightweight, source-to-source compilation system for incrementalizing MCMC updates in probabilistic programs. We have described how C3's two main components, continuations and callsite caching, allow it both to avoid re-executing function calls and to terminate re-execution early. Our experimental results show that C3 can provide orders-of-magnitude speedups over previous lightweight inference systems on typical generative models. It even enables constant-time updates in some cases where previous systems required linear time. We also demonstrate that C3 improves performance by nearly 10x on a complex, compute-heavy inverse procedural modeling problem. Our implementation of C3 is freely available as part of the open-source WebPPL probabilistic programming language.

C3 provides the most benefit for models where the number of latent variables grows with the dataset size. For models with a small number of global latent variables, C3 will not provide any speedup, and in fact the cache overhead may result in a small constant factor slowdown (though adaptation will remove almost all cache lookups in such cases). This is less a limitation of C3 and more an intrinsic expense of such models: any MH implementation will have to completely re-execute on each proposal.

For the Bayesian data analysis models we showed in this chapter, much of C3's performance boost comes from its ability to terminate execution early, i.e. achieving constant-time updates for the HMM program. However, programs for which early

termination is not possible can still see significant performance benefits. For example, when the program involves extensive recursive branching, as in the procedural tree program of Section 3.4, C3 can prune large sub-trees from the overall execution trace.

Chapter 4

Exploring Tightly-Constrained Design Spaces

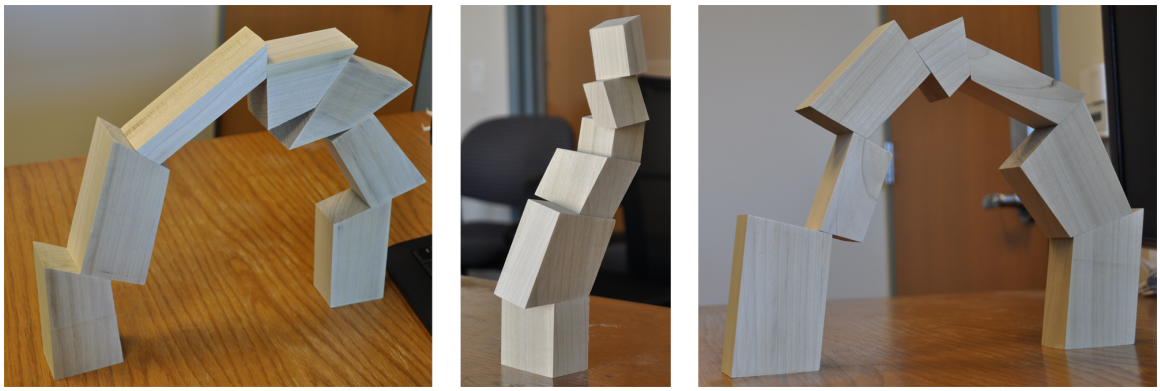


Figure 4.1: Physical realizations of stable structures generated by our system. To create these structures, we write programs that generate random structures (e.g. a random tower or a randomly-perturbed arch), constrain the output of the program to be near static equilibrium, and then sample from the constrained output space using Hamiltonian Monte Carlo.

Considering multiple possibilities is critical in design. Exposure to different examples facilitates creativity—for instance, prototyping multiple alternatives can lead to better-quality final designs [56, 22]. Exploring the whole space of creative options

seems to help people avoid fixation and overcome their unconscious biases [49]. Computation can assist with this exploration by generating suggestions: given a model of the design space, computers can synthesize examples that their users might never have thought of independently. As we argue throughout this thesis, modeling the design space with probabilistic programs provides a powerful combination of generative logic plus constraints.

Real design applications feature a range of constraints, from vague preferences that loosely shape the design space ("Make this object a reddish color") to strict requirements that eliminate entire regions of the space as undesirable ("This container must hold one liter of liquid, up to manufacturing tolerance"). But the tighter these constraints, the more ill-conditioned the underlying probability distribution becomes. As we will show, basic random walk MCMC methods (such as Lightweight MH and C3) break down when faced with tight constraints, especially in high-dimensional design spaces. To work around this problem, developers can implement complex, application-specific MCMC algorithms that exploit knowledge of constraint structure [48, 98]. This strategy does not scale, however, as it requires new algorithms be developed for each new application. General-purpose solutions would be preferable, especially for use with probabilistic programming.

In this chapter, we take a step toward solving this problem by adopting a different sampling algorithm: Hamiltonian Monte Carlo (HMC). HMC is used in Bayesian statistics to train predictive models with many parameters [76]. It excels when the posterior distribution of the parameters given training data causes some parameters to become highly correlated—the same statistical problem as design variables being strongly coupled by tight constraints. Its performance comes from using the *gradient* of the probability distribution to take less-random walks through the state space. This gradient can be computed automatically, making HMC a general-purpose, application-agonistic tool. HMC operates on continuous design domains (i.e subsets of $\mathbb{R}^n$). This property makes it a tool well-suited to graphics applications, since they often feature many continuous quantities (positions, directions, dimensions, colors, etc.)

To evaluate the usefulness of HMC for design suggestion tasks, we implemented the

algorithm in Quicksand, an open-source probabilistic programming language embedded in the Terra language for high-performance computing [89, 18]. We then use our implementation to generate suggestions for two different example applications: vector art coloring and designing stacking structures. These applications employ several challenging and generally-useful constraints, such as physical stability and symmetry. We compare the performance of HMC to classical random walk MCMC on these two examples, demonstrating that HMC provides both qualitatively and quantitatively better design space exploration in the presence of tight constraints.

4.1 Related Work

4.1.1 Design Space Exploration

Design space exploration in computer graphics can be traced back at least as far as the seminal work on Design Galleries by Marks and colleagues [66]. Exploration can be divided into two phases: generating suggestions and navigating between those suggestions. Our work focuses on generating suggestions; other researchers have examined the navigation problem [3, 113].

Researchers have experimented with different algorithmic frameworks for generating design suggestions, including genetic algorithms [124], nonlinear manifold exploration [127], and probabilistic inference [46, 112, 69]. Our system uses probabilistic inference, and the particular inference algorithm it relies on, Hamiltonian Monte Carlo, shares some mathematical similarities with manifold exploration methods.

Design domains can contain discrete variables, continuous variables, or some combination of both. Several existing design suggestion methods operate on purely discrete design spaces, including shape generation by part combination [50, 47] and tiled pattern synthesis [128]. In contrast, our work focuses on continuous design spaces, which are often used to model quantities such as positions, directions, sizes, and colors. In the mixed discrete/continuous regime, an important subclass of design spaces are those where discrete choices dictate the structure of a continuous parameter set [129, 27]. The techniques presented in this chapter can be also applied to the

continuous subsets of these domains, for a fixed setting of the discrete choices.

Probability distributions over design spaces are typically complex, and researchers have explored techniques to make sampling from them more tractable. Parallel tempering, which assists samplers when probability mass is concentrated around multiple modes, is one notable example [112, 69, 62]. In contrast, the techniques we present help when probability mass is concentrated along thin manifolds. The two methods can be used in concert if a design space exhibits both multi-modality and manifold structure.

4.1.2 HMC Applications

HMC has been applied in other areas that require searching through complex design spaces. It has found use in trajectory optimization for robot motion planning [134]. It has also been applied in 3D printing for estimating and correcting material shrinkage during the printing process [41]. Both of these efforts are concerned with optimization problems: they attempt to find the best possible solution in a large design space. In contrast, we seek to explore large sets of possibilites in design spaces.

HMC also been applied to probabilistic programs. One such effort implements HMC in the Church programming language by viewing the gradient computation as a non-standard interpretation of the program [118]. The Stan inference system also uses a variant of HMC to perform inference in user-programmable generative models [**?**]. For experimenting with graphics applications, we chose to implement HMC in Quicksand [89]. Quicksand generates high-performance, low-level code (whereas Church is a high-level, functional language) and is a general-purpose programming language (whereas Stan uses a statistical modeling domain-specific language)—these properties make it easier to efficiently express graphics programs.

4.2 The Problem: Tight Constraints

To illustrate the problem posed by tight constraints, we examine a token example application, constraining the position of a 2D point, that evokes the kind of constraints

that can arise in spatial layout tasks.

Suppose we constrain the position of a point (x, y) with the following energy penalty:

$$\texttt{softeq}(y^4 - y^2 + x^2 - 0.25, 0, \sigma) \tag{4.1}$$

Here, the 'soft equality' function $\texttt{softeq}(z, \mu, \sigma)$ is an alias for the log of the normal distribution with mean μ and variance σ^2 evaluated at z (i.e. $\log \mathcal{N}(z, \mu, \sigma)$). In this case, it penalizes points that fall too far from the 0-isocontour of the function $y^4 - y^2 + x^2 - 0.25$. The bandwidth σ controls the tightness of this factor, or how aggressively it applies its penalty. The top left of Figure 4.2 shows the probability density $\pi(x, y)$ that results from setting $\sigma = 0.1$.

To sample from such a distribution, we could use MH: take some initial current state (x_0, y_0), *propose* a new state $(\tilde{x}_0, \tilde{y}_0)$, and then *accept* that state if its probability does not decrease by too much. If the proposed state is accepted, it becomes the new current state (x_1, y_1), and the process repeats. A simple, popular choice of proposal strategy is to construct $(\tilde{x}_0, \tilde{y}_0)$ by choosing one of x_0 or y_0 at random and making a small random perturbation to it. The top middle of Figure 4.2 shows 2000 samples drawn from $\pi(x, y)$ using MH. The samples form a good approximation to the true distribution.

The same does not hold when the constraint is tightened by decreasing σ to 0.005. The bottom left of Figure 4.2 shows the new probability density $\pi'(x, y)$; the narrow ridges of high probability reflect the tightened constraint. Running MH for the same number of samples on this new distribution gives the result in the bottom middle of Figure 4.2. While MH finds its way to a high-probability region (the phase of sampling statisticians call *burn-in*), it does not fully explore the distribution. Most random perturbations push the point (x, y) off the narrow, high probability ridges, so the perturbation size must be made very small. We also color sample points by time; the spatially-contiguous regions of constant color illustrate the sampler's slow progress. This is a two-dimensional example chosen for ease of visualization; we could exploit this low-dimensionality and our knowledge of the distribution's symmetry to do better via brute force. Unfortunately, this isn't possible for high-dimensional distributions

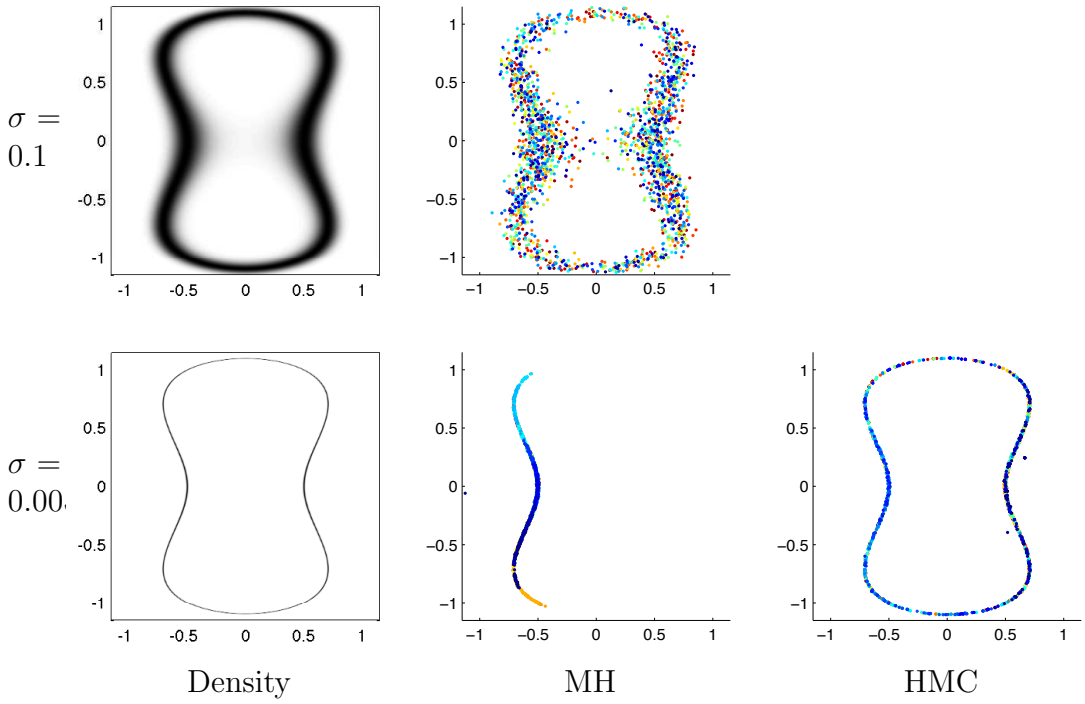


Figure 4.2: Tight constraints in action on a simple 2D example. Top left: The probability density of Equation 4.1 with $\sigma = 0.1$. Top middle: Samples drawn from this density using MH. Bottom left: The probability density of Equation 4.1 with $\sigma = 0.005$. Bottom middle: Samples drawn from this density using MH. Bottom right: Samples drawn from this density using HMC. HMC fully explores the distribution when constraints are tight, while MH does not. Samples are colored by time to illustrate the dynamics of the two algorithms.

with unknown shape, where the variable-coupling problem becomes even worse for MH [76].

We can quantify MH's poor performance using *autocorrelation*, a measure of how similar successive samples are to one another. This is a standard test for assessing MCMC performance for Bayesian statistics [51]. The green line in Figure 4.3 shows the autocorrelation plot of the MH sampling trace, which is far from the ideal value of zero: since the sampler is stuck in the same part of the state space, many samples are similar, so autocorrelation remains high.

Hamiltonian Monte Carlo performs both qualitatively and quantitatively better

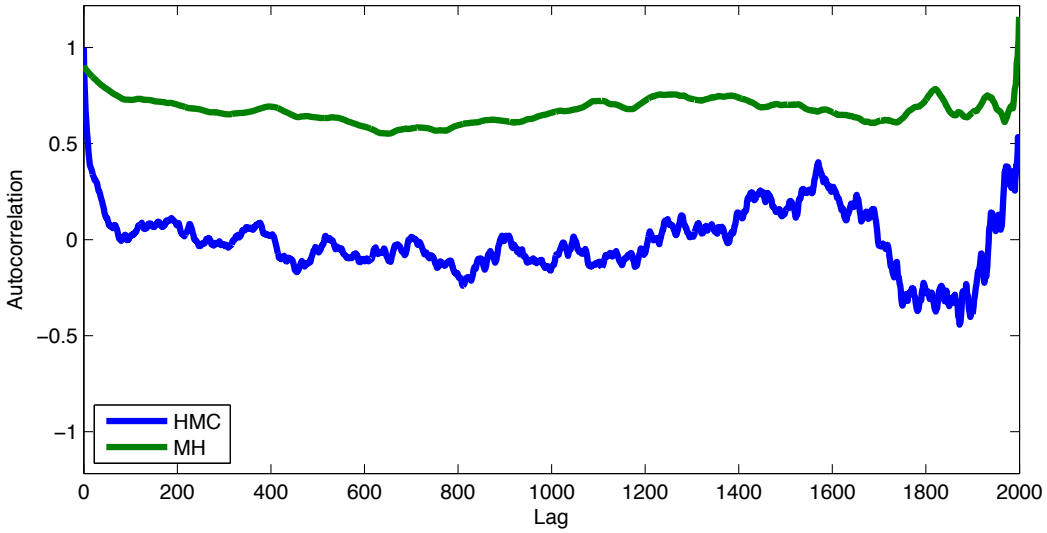


Figure 4.3: Autocorrelation plots for the samples show in the bottom row of Figure 4.2. HMC oscillates around zero (the ideal value), while MH never approaches this target.

on this example. The bottom right of Figure 4.2 shows samples drawn from $\pi'(x, y)$ by HMC given the same computational budget as MH. Visually, these samples represent the distribution much better, and autocorrelation quickly drops to near-zero (Figure 4.3, blue line).

The way HMC works can be explained by physical analogy. If we invert the probability density landscape in the bottom left of Figure 4.2, the thin ridges become narrow valleys. Imagine placing a ball in one of these valleys and rolling it in some random direction. It would roll up and down the surrounding walls, but it would also make progress down the length of the valley. This is the core process underlying Hamiltonian Monte Carlo: it runs a simulation of frictionless Hamiltonian dynamics using the negative log probability $-\log \pi'(x, y)$ as its potential energy.

In the next section, we describe the Hamiltonian Monte Carlo algorithm and our implementation of it in more detail. We then evaluate our system on two different design suggestion tasks: coloring vector art, and designing stable stacking structures (Section 4.4).

4.3 Hamiltonian Monte Carlo

As shown in the previous section, using MH can result in a sampler that moves very slowly across the state space—producing highly-correlated samples—when multiple variables are strongly coupled by tight constraints.

Hamiltonian Monte Carlo (HMC) is a variant of MCMC that can efficiently explore highly-coupled, high-dimensional continuous distributions. It was originally developed for lattice field theory simulations in statistical physics [23], but has since seen increasing adoption in the Bayesian statistics community (see Neal [76] for an excellent overview and survey).

HMC derives its name from Hamiltonian dynamics, which it uses to generate proposals. For this purpose, Hamiltonian dynamics specify the behavior of a frictionless, unit-mass particle with some position $\mathbf{x}$ and momentum $\mathbf{p}$. At a given point in time, the particle has kinetic energy $K(\mathbf{p}) = \mathbf{p}^T\mathbf{p}/2$ and potential energy $U(\mathbf{x}) = -\log \pi(\mathbf{x})$, the sum of which is called the Hamiltonian: $H(\mathbf{x}, \mathbf{p}) = K(\mathbf{p}) + U(\mathbf{x})$.

Given a current state $\mathbf{x}$ from state space $\mathbf{X}$, HMC generates proposals as follows:

1. Sample a random momentum $\mathbf{p} \sim \mathcal{N}(\cdot, 0, \mathbb{I}^n)$.

2. Simulate the dynamics of the particle $(\mathbf{x}, \mathbf{p})$ for L time steps, resulting in the particle $(\mathbf{x}', \mathbf{p}')$.

3. Accept the new particle with probability $\min[1, \exp(H(\mathbf{x}, \mathbf{p}) - H(\mathbf{x}', \mathbf{p}'))]$.

Essentially, HMC performs a Metropolis Hastings propose + accept step on an augmented state space where the states are $(\mathbf{x}, \mathbf{p}) \in \mathbb{R}^n \times \mathbb{R}^n$. Instead of walking through the state space with single steps in random directions, HMC follows long, multi-step paths along the energy landscape defined by $-\log \pi(\mathbf{x})$. When this landscape is defined by constraints, as in our applications, adhering to its contours corresponds to constraint satisfaction.

To perform Step 2 above, we must simulate the time evolution of our fictitious

particle. This is governed by the differential equations:

$$\frac{d\mathbf{x}}{dt} = \nabla_{\mathbf{p}} H(\mathbf{x}, \mathbf{p}) = \nabla K(\mathbf{p}) = \mathbf{p}$$
$$\frac{d\mathbf{p}}{dt} = -\nabla_{\mathbf{x}} H(\mathbf{x}, \mathbf{p}) = -\nabla U(\mathbf{x}) = \nabla \log \pi(\mathbf{x})$$

which are numerically simulated using the discrete-time update rules

$$\mathbf{p}(t + \frac{\epsilon}{2}) = \mathbf{p}(t) + \frac{\epsilon}{2} \nabla \log \pi(\mathbf{x}(t))$$
$$\mathbf{x}(t + \epsilon) = \mathbf{x}(t) + \epsilon \mathbf{p}(t + \frac{\epsilon}{2})$$
$$\mathbf{p}(t + \epsilon) = \mathbf{p}(t + \frac{\epsilon}{2}) + \frac{\epsilon}{2} \nabla \log \pi(\mathbf{x}(t + \epsilon))$$

known as the *leapfrog* integrator [57]. The leapfrog scheme has two critical properties that make it work for HMC proposals. First, it is a *symplectic* integrator (i.e. the map from $\mathbf{X}$ to $\mathbf{X}$ that it defines preserves volume). Second, it is *time-reversible*: if $\texttt{leapfrog}((\mathbf{x}, \mathbf{p}), \epsilon) = (\mathbf{x}', \mathbf{p}')$, then $\texttt{leapfrog}((\mathbf{x}', -\mathbf{p}'), \epsilon) = (\mathbf{x}, -\mathbf{p})$. In other words, flipping the direction of momentum and simulating 'backwards' returns the system to the state from which it started. These properties are key to proving that HMC satisfies the detailed balance condition and thus defines a valid MCMC sampler [76].

Parameters HMC has two parameters: the number of leapfrog steps L and the simulation step size ϵ. The tighter the constraints used in a particular application, the smaller ϵ must be to keep the Hamiltonian dynamics simulation numerically stable. Consequently, the number of steps L must increase for HMC proposals to make progress exploring the state space. We use a method proposed by Hoffman and Gelman [40] to automatically set ϵ and leave L as the only free parameter in the system. We found $L = 100$ to be sufficient for most of our experiments. There is also a variant of HMC that attempts to automatically, adaptively determine L as it traverses the state space [40].

Bounded variables Many design applications require variables with strict bounds (e.g. "this object must be between 10 and 50 cm long"). These can be incorporated into HMC via *variable transformation*: letting a variable x be unbounded, but transforming it such that the value $\tilde{x}$ exposed to the program is bounded. For a variable with both a lower bound l and an upper bound u, the typical transformation is logistic:

$$\tilde{x} = l + (u - l) \cdot \frac{1}{1 + \exp(-x)}$$

Similar transforms exist for one-sided bounds [104].

Implementation We implemented Hamiltonian Monte Carlo in Quicksand, a probabilistic programming language embedded in Terra [89, 18]. We chose this implementation target because Terra is a low-level language that compiles to efficient machine code, which we believe to be important for achieving sufficient performance for graphics applications. We implement HMC as a custom MCMC kernel in Quicksand. Quicksand supports inference over arbitrary programs, including recursive programs and programs whose set of random choices may change based on control flow decisions. Since HMC operates on $\mathbb{R}^n$, we can only use HMC to explore parts of the execution space with a fixed set of random choices. HMC could be composed with other Quicksand MCMC kernels (such as LARJ-MCMC [129]) to perform inference on structure-changing programs.

Our system uses automatic differentiation (AD) to compute the gradients required by HMC, relieving the user of having to derive gradients manually. In particular, it uses *reverse-mode* AD, which computes the gradient in just two passes over the program, regardless of state space dimensionality [14, 102]. Efficient symbolic differentiation could also be used for some parts of the program and might further improve performance [34].

4.4 Evaluation

We use our implementation to evaluate the usefulness of HMC for computational design by generating suggestions for two example applications: vector art coloring and building stacking structures. These are two unrelated application domains that both require tight constraints to eliminate undesirable regions of the design space.

We compare the statistical efficiency of HMC with MH and show that HMC's improved efficiency leads to qualitatively better suggestion results. For fair comparison, we initialize each algorithm by burning in for a fixed number of MH iterations. In all experiments, MH proposal bandwidths are automatically adapted to give $\sim 23\%$ acceptance, and HMC steps sizes are automatically adapted to give $\sim 65\%$ acceptance. Selecting a good target acceptance rate can be highly problem-specific, but there is theoretical evidence that these are good general settings for their respective algorithms [94]. Each algorithm is allotted the same computational budget in terms of *program evaluations*. An HMC sampler with L leapfrog steps uses $2L$ as many evaluations to generate a sample as an MH sampler (the factor of 2 comes from the reverse-mode AD backwards pass). So if the HMC sampler runs for 1000 iterations using 100 leapfrog steps, MH is allowed to run for $(2 \cdot 100) \cdot 1000 = 200000$ iterations.

We also collect timing data to demonstrate that our implementation generates suggestions quickly enough for practical use. All timing information reported in the following experiments was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.8.5.

Finally, source code for these examples is available on GitHub at `https://github.com/dritchie/graphics-hmc`.

4.4.1 Vector Art Coloring

In vector illustrations, a significant portion of a design's visual impact comes from color choice. Designers must consider semantics as well as aesthetics to create plausible and harmonious colorings. For example, certain objects may be strongly associated with specific colors (e.g. sky to blue), regions that are part of the same material

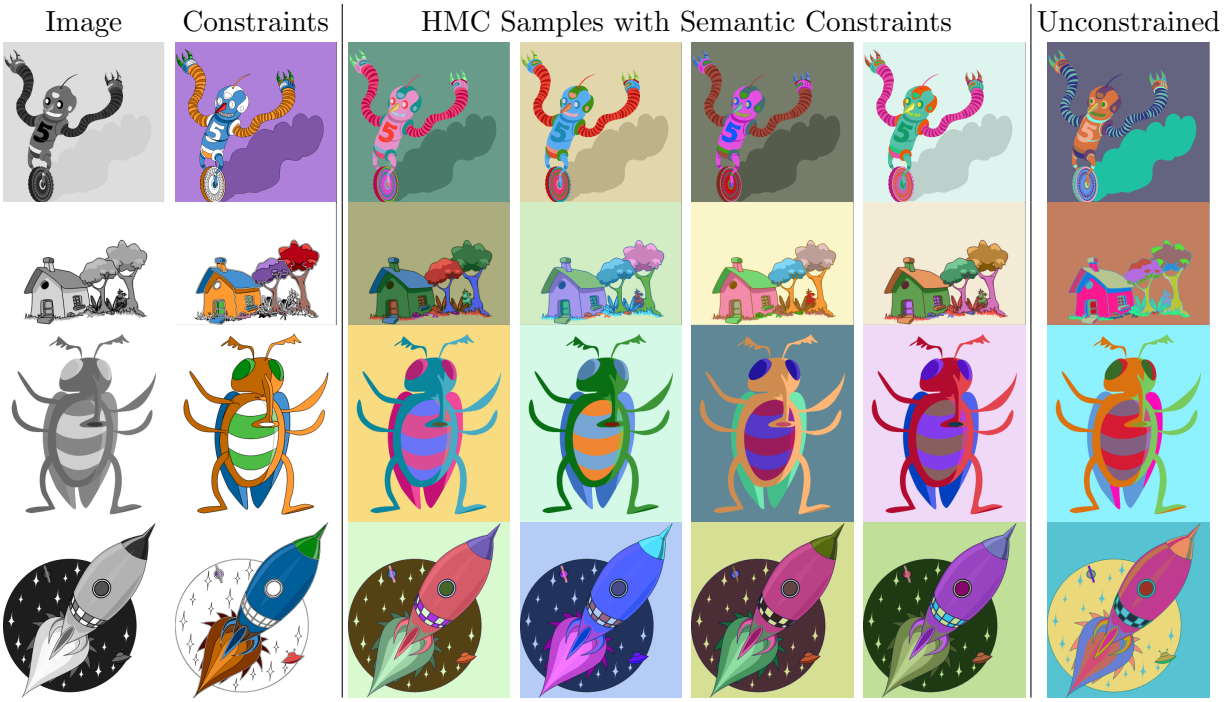


Figure 4.4: Vector art colorings with and without semantic constraints. Image: The image template, which maps individually-recolorable regions to different grayscale levels. Constraints: Visualization of the applied constraints. Same-Chroma constraints over regions are visualized with the same hue. White regions have no hue constraints. Lightness-Relation constraints for regions of the same hue are visualized with darker or lighter shades. *Additional Lightness-Relation constraints are as follows*: *Robot*: eye centers lighter than helmet lights, helmet lights lighter than helmet and robot body, number "5" darker than body. *House*: sky lighter than roof and tree highlights, lineart darker than shadows. *Rocket*: lineart darker than space, stars lighter than middle flame, window darker than rocket body.

may need to have similar hues, and shading effects may dictate that some regions should be lighter than others.

Previous work has modeled the compatibility of color combinations and arrangements [78, 77, 62]. For pattern images, Lin and colleagues introduce a coloring model composed of soft constraints learned from artist examples [62]. Their system uses MH, augmented with variable swaps and parallel tempering for faster exploration of multiple modes, to sample coloring suggestions from the learned model.

To compare the effectiveness of HMC to MH for coloring constrained vector art,

we add tight semantic constraints to a simplified version of the coloring model by Lin and colleagues. Specifically, we add *Same-Chroma* constraints, which enforce that two regions should have the same chromatic content (i.e. the same color irrespective of lightness), and *Lightness-Relation* constraints, which enforce that one region should be brighter or darker than another. These constraints are much tighter than the soft constraints that comprise the base model, and thus they are likely to cause trouble for MH. Refer to Appendix B for the full specification of our coloring model.

Figure 4.4 shows examples of sampling from three vector art images using HMC under multiple Same-Chroma and Lightness-Relation constraints. The first and second columns of the figure show the vector art template and a visualization of the semantic constraints applied. Under these tight constraints, HMC is still able to sample a variety of different colorings.

In Figure 4.5, we compare the performance of HMC and MH under the same computational budget. We ran the HMC sampler for 1000 iterations using 100 leapfrog steps and the MH sampler for the equivalent of 1000 HMC iterations (200000 iterations). The first two columns show 'coverage maps' for the two sampling traces, where stronger blue regions indicate that more colors were sampled for that region and that the sampler is exploring the space better. To compute coverage, we discretize CIELAB space into 256 bins (4x8x8) and count the percentage of bins visited for each region. HMC consistently samples more colors than MH. The background in the Bug example has high coverage under MH because it does not participate in any semantic constraints. The third column shows autocorrelation plots for the runs, again demonstrating that the MH samples are more self-similar.

Timings for these examples are shown in Figure 4.6. We report the time consumed by the burn-in phase (the 'start-up cost' of the system), the time taken by sampling (when the system is generating useful suggestions) as well as the acceptance ratio of the HMC sampler. The HMC sampler draws around 60 new samples per second (number of samples drawn multiplied by acceptance rate and divided by sampling time). Rates such as these should be sufficient for use in an interactive coloring tool.

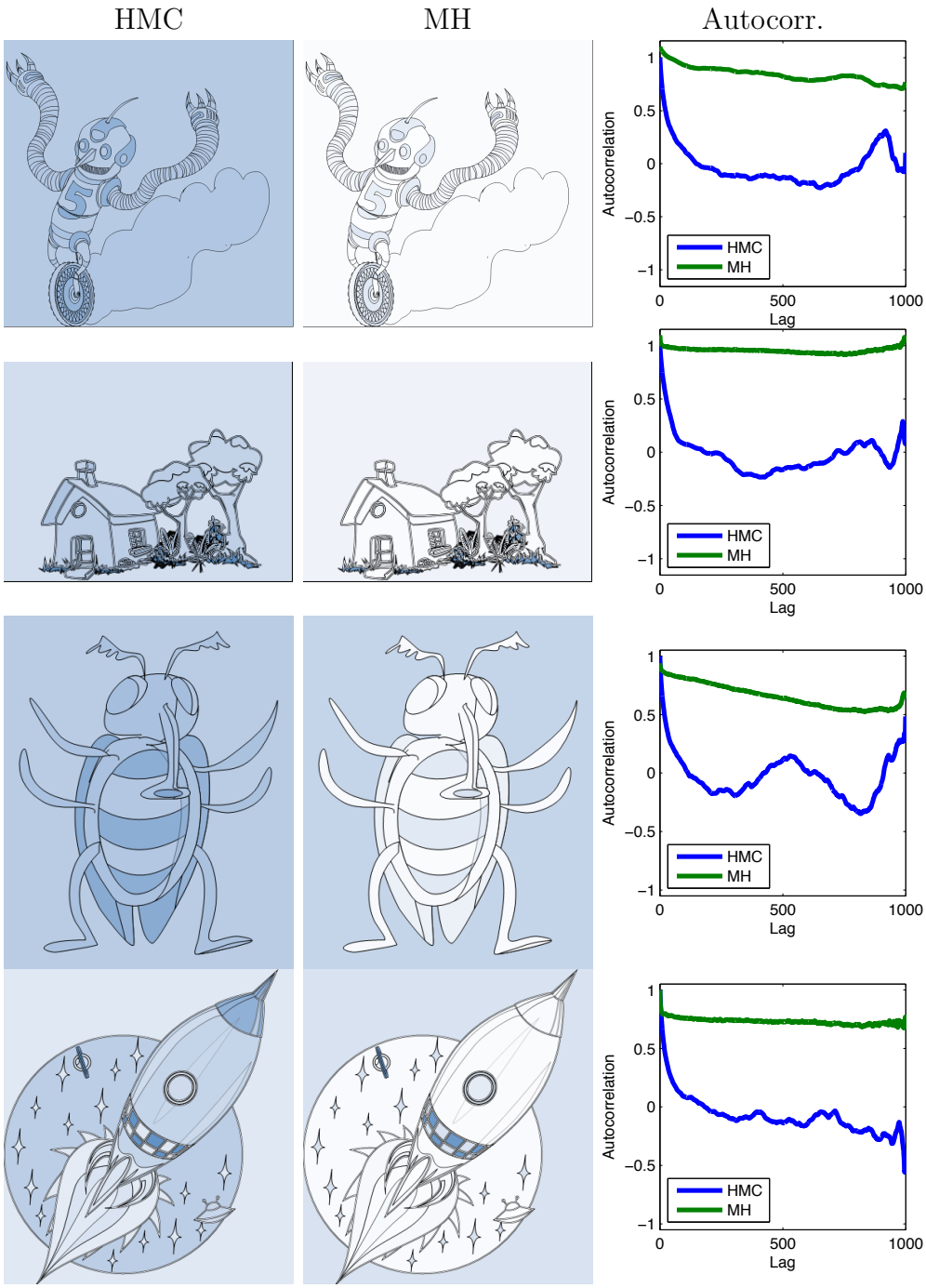


Figure 4.5: The first two columns show coverage plots for HMC and MH sampling on the three image templates. Darker shades of blue indicate that more colors were sampled for the given region, while white indicates fewer colors sampled. Colors are counted by discretizing CIELAB space into 256 bins. The last column shows autocorrelation plots comparing HMC and MH.

Example	$\|\mathbf{X}\|$	Burn-in	Sampling	Accept. ratio
Robot (MH)	57	0.12 s	12.66 s	0.24
Robot (HMC)	57	0.12 s	10.61 s	0.65
House (MH)	60	0.13 s	13.38 s	0.24
House (HMC)	60	0.13 s	10.67 s	0.65
Bug (MH)	30	0.06 s	6.08 s	0.24
Bug (HMC)	30	0.06 s	3.56 s	0.63
Rocket (MH)	60	0.12 s	14.53 s	0.23
Rocket (HMC)	60	0.12 s	10.61 s	0.63

Figure 4.6: Timing data for the examples shown in Figure 4.4. $|\mathbf{X}|$ is the number of random choices made by a program.

4.4.2 Stable Stacking Structures

People are fascinated with the stability of physical structures. The Leaning Tower of Pisa draws over a million visitors each year, games such as Hasbro's Jenga and Areaware's Balancing Blocks have enduring popularity, and balancing rock sculptures have become a form of performance art (Figure 4.7). In this section, we consider the computational design of stacking structures made of rigid blocks that remain stable despite their apparent precariousness.

Prior work has addressed the stability of design artifacts in domains such as truss structure design, 3D printing, and procedural building grammars [101, 83, 116]. These projects pose stability as an optimization problem: given an initial input object (e.g. a 3d model or procedural grammar derivation), seek toward a configuration of the object that is stable. In contrast, we wish to explore the variety of possible configurations of a given structure that will stand.

For a rigid structure to be stable, it must be in *static equilibrium*: the net force and net torque on every component must be zero. In general, these forces are not directly computable from the structure's geometry but are defined implicitly by this equilibrium condition. We can think of them as random variables whose values are

Figure 4.7: Real-world inspiration for our stable stacking application. Left: Areaware's Balancing Blocks game. Right: Balancing rock sculpture.

tightly coupled by the equilibrium constraint. This suggests a simple generative model for creating stable structures: generate a random block structure, introduce latent variables representing forces between blocks, encourage equilibrium with a tight constraint, and sample from the resulting distribution using HMC. See Appendix B for the full specification of our statics model.

To keep our example application simple, we consider only convex, hexahedral blocks. While this simplification does not capture all the rich detail of stacking structures in the real world (e.g. the irregular convex polyhedra in Figure 4.7, left), it admits a wide range of stacking arrangements.

Figure 4.8 shows some examples of sampling from a random block stacking program using both HMC and MH. We show three interesting structures sampled by HMC, as well as the 'average' structure produced by both algorithms. We also compare the autocorrelation curves of the two sample traces, where autocorrelation is computed by reducing each structure to a vector of all block vertex positions. We

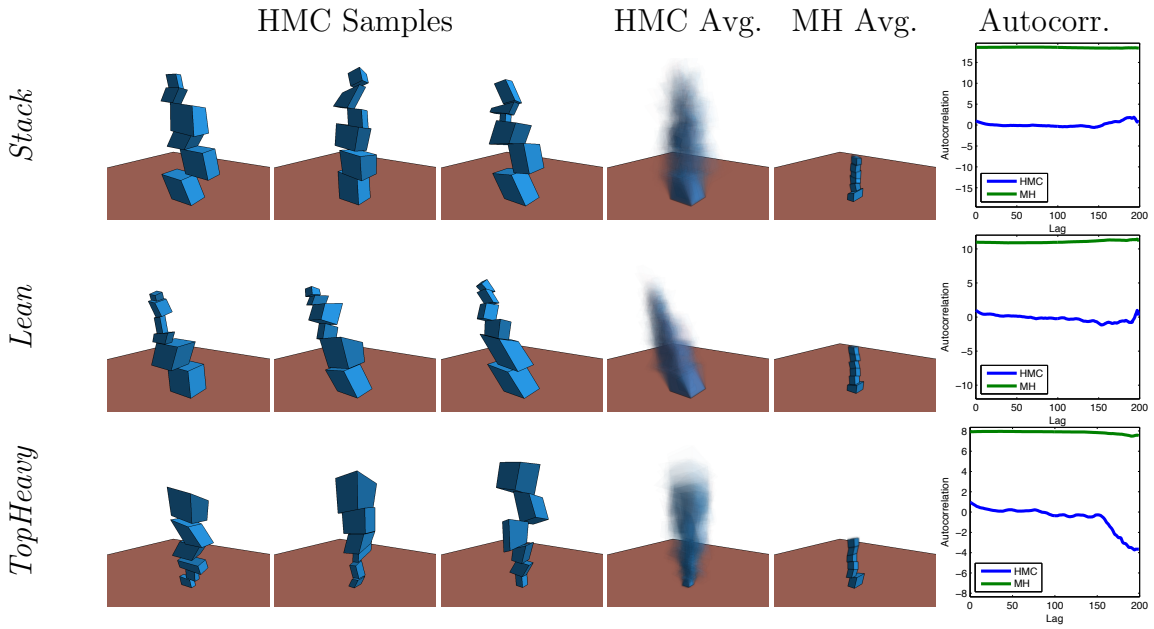


Figure 4.8: Generating stable block stacks with different criteria. Top: A stack with no additional constraints. Middle: Encouraging the stack to lean in a particular direction. Bottom: Encouraging each block to be twice as large as the block below it. For each scenario, we show three HMC samples, the average of all samples generated by each method (200 for HMC, 400000 for MH), and a comparison of their autocorrelation curves.

ran the HMC sampler with 1000 leapfrog steps for 200 samples and the MH sampler for the equivalent of 200 HMC samples (400000 iterations).

The top row shows results from the stacking program. In the middle row, we add a factor to encourage the stack to lean in a particular direction by penalizing the distance of each block's center of mass to a target line. We also generate precarious-looking 'top-heavy' stacks by adding a factor that encourages each block's volume to be twice as large as that of the block below it (bottom row). HMC has little trouble exploring the complex probability landscape induced by the stability constraint, but MH struggles, seeking out a local maximum and barely deviating from it. MH generates small structures because we initialize the latent force variables to zero, so it can quickly minimize force and torque residuals by shrinking all blocks to the minimum size allowed by the program.

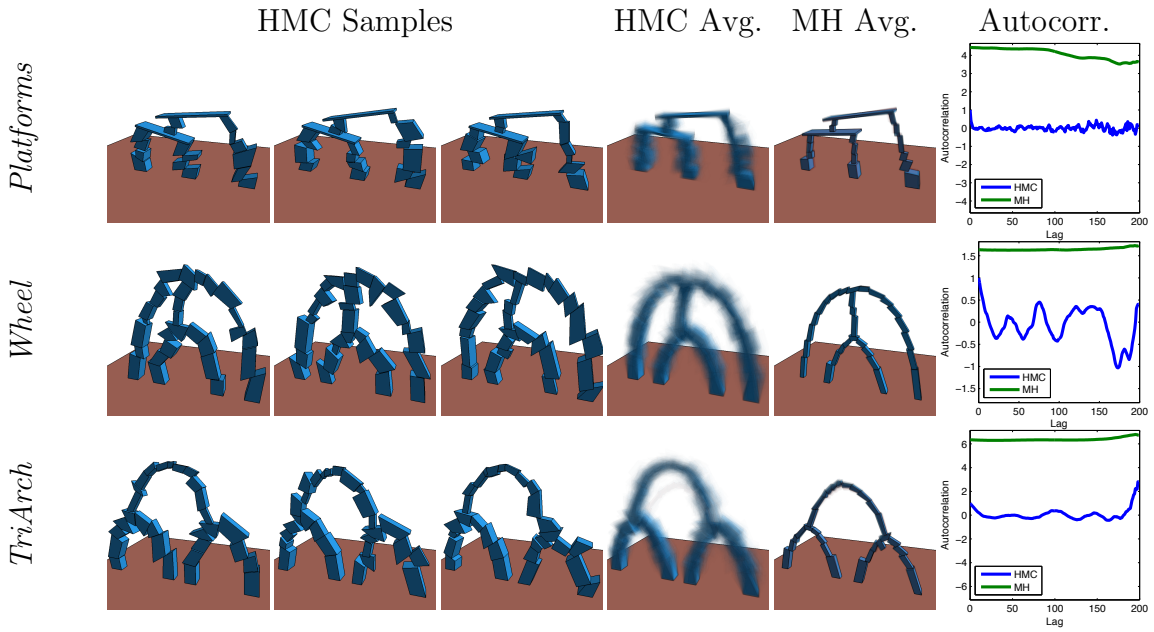


Figure 4.9: Generating stacking structures with more complex, cyclical topologies.

Figure 4.9 shows this same comparison with programs that generate more topologically-complex structures. HMC again successfully samples many interesting configurations of these structures, whereas MH again becomes stuck. The complex, cyclical contact relationships in these examples make the space of stable configurations more tightly constrained than in the examples of Figure 4.8. This video shows animations of some of these sample traces which better illustrate the dynamics of the different algorithms: `https://www.youtube.com/watch?v=Ymtl-w_97sU`.

Since it uses softened constraints, HMC in general cannot guarantee that the structures it samples will be exactly in equilibrium, only that they will be close to it. Thus, these examples need more leapfrog steps (1000) because constraint bandwidths have to be kept very tight to keep the sampler sufficiently close to the static equilibrium manifold (see Appendix B). We used a linear program solver to check whether each generated structure satisfies the equilibrium equations, and nearly all of them do.

Timing statistics for these examples are shown in Figure 4.10. In general, running

Example	$\|\mathbf{X}\|$	Burn-in	Sampling	Accept. ratio
Stack (MH)	118	30.57 s	65.22 s	0.23
Stack (HMC)	118	36.30 s	148.76 s	0.61
Lean (MH)	118	24.10 s	48.39 s	0.23
Lean (HMC)	118	27.90 s	118.86 s	0.62
TopHeavy (MH)	118	33.35 s	58.48 s	0.23
TopHeavy (HMC)	118	38.04 s	186.62 s	0.59
Platforms (MH)	330	62.52 s	118.75 s	0.25
Platforms (HMC)	330	62.54 s	279.9 s	0.59
Wheel (MH)	555	93.76 s	688.70 s	0.26
Wheel (HMC)	555	98.04 s	729.54 s	0.63
TriArch (MH)	555	94.90 s	191.25 s	0.26
TriArch (HMC)	555	95.5 s	700 s	0.62

Figure 4.10: Timing data for the examples shown in Figures 4.8 and 4.9. $|\mathbf{X}|$ is the number of random choices made by a program.

time scales linearly with the complexity of block topology. The sampling rate is lower than in the coloring examples, since the complexity of the static equilibrium constraint necessitates taking more small leapfrog steps. Each step is also more expensive, since the statics model requires more computation and makes many more random choices.

To illustrate another use of tight constraints, Figure 4.11 shows three structures generated from a simple arch program and the *TriArch* program with an additional bilateral symmetry constraint. Adding this constraint takes very little extra effort in our system: the program simply reflects the structure about the symmetry plane and then applies a `softeq` factor to each symmetric pair of block vertices.

To validate our statics model, we built physical prototypes of some structures generated by our system (Figure 4.1). In the program used to generate these examples, we restricted block shapes to be planar extrusions to allow us to easily cut them out of wood stock. All blocks shown were cut from 38 mm ($1\frac{1}{2}$ in) poplar, whose density

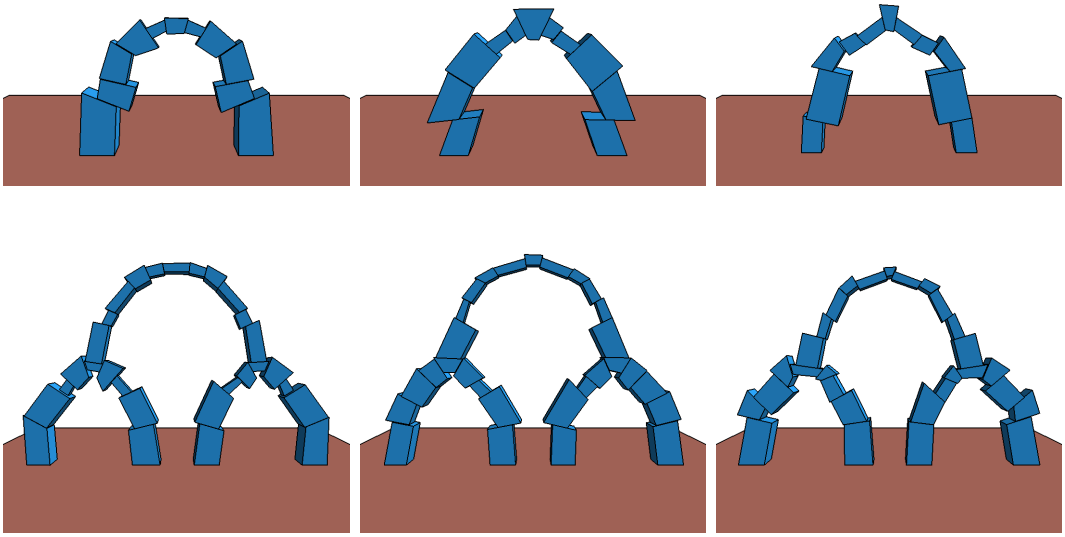


Figure 4.11: Structures generated with an additional constraint encouraging bilateral symmetry.

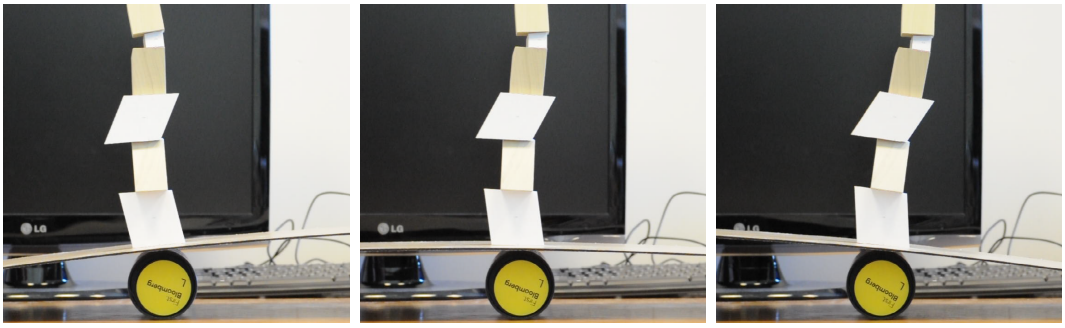


Figure 4.12: Testing a block stack generated under the constraint that it be stable at up to $\pm 10^\circ$ tilts of the ground plane.

and coefficient of friction we estimated as 425 kg/m^3 and 0.3, respectively [5].

To increase the stability of a structure, we can enforce that it be stable under some class of perturbations, rather than at a single rest configuration. Figure 4.12 shows a block stack generated under the constraint that it stand under as much as $\pm 10^\circ$ tilts of its ground plane. To enforce this condition, we write a program that generates a random stack as before, then rotates the entire scene $\pm 10^\circ$ about one axis, applying a stability constraint at each rotation.

The generated structures shown thus far are not necessarily stable at every step

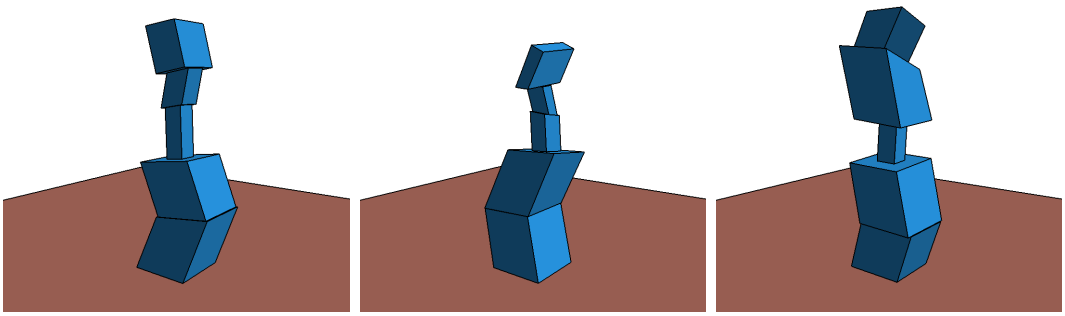


Figure 4.13: Block stacks generated under the additional constraint that they be stable at every intermediate construction step.

of their construction, which can complicate the process of physically building them. We can mitigate this problem by applying a stability factor at intermediate phases of structure generation, as opposed to just one at the end. Figure 4.13 shows three example stacks generated this way. To generate more exciting structures under this constraint, one could write programs that use temporary scaffolding to hold the structure up, treat the presence and configuration of that scaffolding as random variables, and infer plausible build processes. This is a promising avenue for future work.

4.5 Chapter Summary

This chapter introduced Hamiltonian Monte Carlo to probabilistic computational design. We implemented HMC in a high-performance probabilistic programming language, and we evaluated it on two example applications, showing that it can efficiently generate suggestions in highly-constrained scenarios.

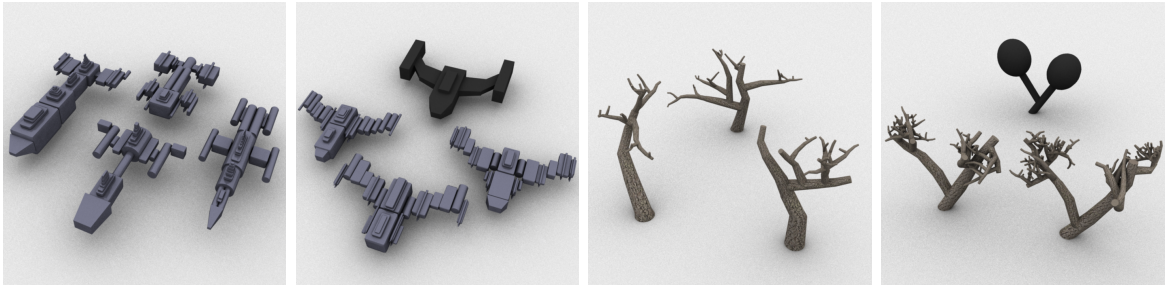
HMC relies on the gradient $\nabla \log \pi(\mathbf{x})$ to make proposals, so the probability π must be continuous and differentiable everywhere. This requirement limits the factors that can be used to define π. For example, `min` and `max` are useful for defining penalty functions but cannot be used directly, though they can often be approximated with relaxed versions. Graphics applications often feature other complex, highly-discontinuous functions, such as rendering and collision. These functions might also be similarly relaxed through *smooth interpretation*, a technique for automatically

deriving a smooth, differentiable approximation of programs [10].

Complex hard constraints also remain challenging; sufficiently tight soft constraints may be acceptable, as in the case of our stacking equilibrium examples, but this will not always be the case. An extension to HMC that explicitly adheres to a manifold may provide a good solution [8]. Complex constraints on discrete variables are also difficult—integrating SAT solvers into MCMC methods has been shown provide some traction here [125].

Chapter 5

Handling Branching Structure with SOSMC



Forward Sampling SOSMC Sampling Forward Sampling SOSMC Sampling

Figure 5.1: Controlling the output of highly-variable procedural modeling programs using our Stochastically-Ordered Sequential Monte Carlo algorithm. Here, the controls encourage volumetric similarity to a target shape (shown in black).

The inference algorithms described in the previous two chapters both belong to the family of Markov Chain Monte Carlo (MCMC) methods. But other Bayesian posterior sampling algorithms are available: another popular choice is Sequential Monte Carlo (SMC). SMC uses a set of samples, or *particles*, to represent a distribution that changes over time as new evidence is observed. As the distribution changes, SMC shifts more particles (and thus more of its computational budget) to higher-probability regions of the state space. For probabilistic models that fit this pattern of

'evidence arriving over time,' such as modeling the location of a mobile robot, SMC is often the method of choice: the incremental evidence it receives provides feedback early and often, allowing it to converge quickly [21]. In contrast, MCMC receives feedback only after running through the entire model.

This 'incremental evidence' pattern could be beneficial for controlling procedural models. Many popular classes of procedural models in graphics, such as tree models, feature deep branching structures. Controlling such a model to, say, produce a particular shape thus requires careful coordination between long chains of random variables.

Can we use Sequential Monte Carlo to control procedural models? Procedural models are typically hierarchical and recursive—we need to cast them instead as sequential processes, where control can be imposed incrementally over time. Fortunately, representing procedural models with probabilistic programs makes this easy, since programs have sequential semantics: they execute in a series of discrete time steps. Control can be imposed incrementally by evaluating a scoring function on the incomplete model at each step, providing an estimate as to how well the algorithm is doing thus far.

However, there are multiple ways to sequentialize a structured procedural modeling program—and as we will show, SMC does not always perform well using the depth-first ordering given by most modern, stack-based programming languages. It is typically not clear *a priori* what the best ordering(s) will be for a given program and control scoring function: in the absence of any special knowledge, a good strategy might be to execute the program in *random* order.

Following this insight, we introduce a new variant of SMC, *Stochastically-Ordered Sequential Monte Carlo (SOSMC)*, in which each particle executes the program in an independent, random order. We also prove that this algorithm is a correct, asymptotically-unbiased sampler for the posterior distribution defined by the constrained program. To implement SOSMC for procedural models expressed as general-purpose probabilistic programs, we also introduce a new programming primitive, the *stochastic future*, whose use requires minimal modification to the original program. We then show that SOSMC can handle a range of procedural models and controls

explored in the literature, and that it generates better-scoring samples under tight time budgets than either normal SMC or Metropolis-Hastings (MH). For small computational budgets, SOSMC's outputs often score nearly twice as high as those of normal SMC or MH.

We give a high-level overview of our main insights and approach in Section 5.2, then we formally describe the SOSMC algorithm in Section 5.3 and our prototype implementation in Section 5.4. In Section 5.5, we evaluate the algorithm's performance on a variety of procedural models with constraints and compare to other sampling methods.

5.1 Related Work

Controlled Procedural Modeling As covered in Section 2.1, several existing projects aim to control procedural models through probabilistic inference. One uses reversible-jump MCMC to direct the output of stochastic context free grammars [112]. Another uses similar MCMC techniques to guide L-system-based trees toward a target polygonal model [106]. Others develop new trans-dimensional MCMC methods to solve complex layout problems [129] or use MCMC to make parameterized models of urban environments satisfy desired criteria [114]. These all use MCMC as their core control algorithm; in contrast, we focus on Sequential Monte Carlo for its potential performance benefits.

There have also been several non-probabilistic approaches to directing procedural models. Environmentally-sensitive L-systems and Open L-systems allow communication between a procedural model and its environment [86, 75]. Another approach decomposes the modeling domain into geometric guides to which the procedural model should adhere [4]. These approaches affect the model as it evolves. Our approach can be thought of as generalizing this type of control using probabilistic inference.

Sequential Monte Carlo Sequential Monte Carlo has a long history, beginning with the simulation of self-avoiding polymer chains [37, 96]. A critical point came with the introduction of a resampling step, allowing the reallocation of particles according

to their probability [31, 108]. The resulting algorithm, called the *bootstrap filter*, was designed for linear time-series processes. We extend it for structured processes by linearizing the process and treating the linearization order as additional set of random variables. It can be shown that, as the number of SMC particles approaches infinity, their distribution approaches the target posterior density [100, 31]. We prove that this distribution is invariant under linearization order, thus re-ordering does not change program semantics.

SMC in Computer Graphics Sequential Monte Carlo has found applications in computer graphics already. It has been applied to Monte Carlo integration for physically-based rendering, in particular rendering with participating media [26, 80]. It has also been used to control virtual characters responding to dynamic environments [36]. These applications have straightforward sequential interpretations: propagation of light along a path through space, or the motion of a character over time. In contrast, we focus on structured procedural models, which have many possible sequentializations.

SMC belongs to the family of *population-based methods*, which evolve a population of samples toward some desired goal. This general approach has also been used for 3D shape design [124]. This system maintains a complete set of shapes at all times, whereas ours works with incomplete shapes defined by partial program executions.

SMC for Probabilistic Programs Sequential Monte Carlo has also previously been applied to probabilistic programs. Anglican implements several SMC methods, including sophisticated SMC/MCMC hybrids [121]. Probabilistic C uses OS multiprocessing primitives to construct efficient, parallel implementations of these same algorithms [79]. It is also possible to implement these algorithms using a continuation-passing-style compiler [30]. These systems are restricted to handling a fixed number of time steps—the common case in statistical inference, where each step corresponds to a data point. In contrast, we are concerned with scenarios that have a variable number of steps, as this situation arises often with structured, recursive procedural models.

5.2 Approach

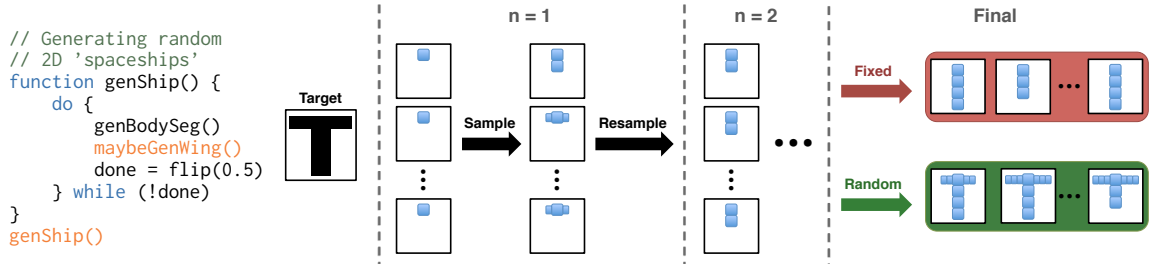


Figure 5.2: *(a)* A program that generates simple random spaceships. Orange-highlighted function calls can be executed in any order with respect to one another. *(b)* SMC resampling favors higher-scoring states, so particles that fill in the body first will dominate. Under fixed ordering, particles skip wing generation altogether, whereas random ordering can defer wing generation until after body generation.

In this chapter, we focus on programs that generate models through hierarchical, recursive accumulation of geometric primitives into an implicit global state. To illustrate our approach, we use an example program that generates random simplified spaceships out of blocks (Figure 5.2a). The program generates the ship body by placing a random number of contiguous blocks and may randomly grow wings, also made of a random number of blocks, from any body segment. For brevity, we do not show pseudocode for the wing-creation function `maybeGenWing`—its structure is similar to that of `genShip`. We will use SMC to sample from this program under a scoring function that encourages similarity to a target shape.

SMC runs N copies of the program, called *particles*, (conceptually) in parallel. Particles execute until they arrive at a barrier synchronization point—this is the *sampling* phase. In our procedural modeling programs, barriers occur when programs generate a new geometric primitive. SMC computes the score of each particle and then randomly samples N particles in proportion to their scores: high-scoring particles are sampled more often, and low-scoring ones are sampled less often, or not at all. This is the *resampling* phase, and these new particles become the input for the next sampling phase. Resampling ensures that the algorithm concentrates particles (and

Algorithm 1 SMC for procedural modeling programs

```
procedure SMC(program, scorefn, N)
    P ← N new particles (instances of program)
    W ← N real-valued weights
    while some particle p ∈ P has not terminated do
        // Sample
        for all unterminated particles p ∈ P do
            Run p until it generates a new geometric primitive
        // Score
        for i = 1 to N do
            W(i) ← scorefn(P(i))
        NORMALIZE(W)
        // Resample
        P ← WEIGHTEDSAMPLEN(P, W, N)
```

thus its computational budget) in high-scoring regions of the state space. Essentially, SMC operates like a stochastic version of beam search [6]. Algorithm 1 shows high-level pseudocode for running SMC on procedural modeling programs.

The first column of Figure 5.2b shows a hypothetical set of particles that have passed the first barrier—that is, they have placed one primitive, which in this case must be a body segment. At the next barrier (second column), some particles will randomly start growing wings from the first body segment, while others will instead proceed with the next body segment. Because body segments are larger than wing segments, placing a body segment brings the model closer to the target more quickly than placing a wing segment. Thus, the resampling phase will favor particles that place body segments over those that place wing segments—body-segment particles will dominate the next round (third column).

Consider the calls to `genShip` and `maybeGenWing`, highlighted in orange in Figure 5.2a. These calls generate independent components of the model and could in principle interleave their execution in any order with respect to one another. However, most programming languages will execute them in a fixed, depth-first order, which causes a problem in this example: all of the second-round particles decided not to generate wings on the first body segment, and SMC has no mechanism to reverse that decision.

The best possible result from this point on are ships with bodies that match the target, but no wings (Figure 5.2b, red box). We could try to eliminate this problem by only resampling after body segment generation, but this would leave wing generation without any guidance from resampling, requiring it to match the target by pure chance. And even if this fix worked, it would be specific to this program—we seek general-purpose solutions that work for any procedural model.

Now suppose each particle executes the calls to `genShip` and `maybeGenWing` in a random order. This means that some second-round particles likely deferred execution of `maybeGenWing` and instead continued executing `genShip`—they have yet to decide if they will generate wings on the first body segment. By deferring this decision, SMC can generate results that have both body and wings that match the target (Figure 5.2b, green box). Sequential Monte Carlo is a form of importance sampling, and here execution order randomization helps it sample the most important objects first.

These execution-order-sensitive situations do not occur in the linear time-series models for which SMC was developed, but they frequently arise in structured procedural modeling. It is clear that some orderings are better than others, but it is *not* always clear which orderings those are. Even if known, it is cumbersome to explicitly express specific orderings in the program text. And finally, the best orderings depend upon the score function being used—it is unreasonable to expect the programmer to restructure her code for each new control imposed on a model.

This is the motivation behind Stochastically-Ordered Sequential Monte Carlo: since we cannot know what execution orderings are good *a priori*, we randomize them, in the hope that randomization will discover good orderings on average. After formally describing SOSMC and its implementation in Sections 5.3 and 5.4, we show that randomization does lead to reliably better results, both qualitatively and quantitatively (Section 5.5).

5.3 SOSMC

Having outlined our approach intuitively, we now formally define the probability distribution sampled by the SOSMC algorithm. SMC algorithms are specified as

sampling from a sequence of distributions p_1, p_2, ...; the final distribution p_N is often the one of interest. These distributions are usually defined over a growing set of variables x, e.g. $p_1(x_1)$, $p_2(x_1, x_2)$, and so on. These variables typically represent states which evolve over time.

Defining such a sequence of distributions for SOSMC is more complicated, as general procedural modeling programs follow a structured execution that does not conform to a single, linear time-series interpretation. Thus, we augment our state space of variables x to include execution ordering choice variables o in addition to the program's own random choice variables r.

We define the intermediate distribution p_n to be the distribution over all execution traces which generate n or fewer geometric primitives. Let $\mathbf{x}_n$ be the sequence of all random choices made up to primitive n, where $\mathbf{x}_0$ is empty. As subsets of this sequence, let $\mathbf{r}_n$ denote the procedural model's random choices, and let $\mathbf{o}_n$ denote the ordering choices. We will sometimes refer to $\mathbf{r}$ as a *trace* through the procedural model program. The intermediate distribution p_n can then be defined recursively as

$$\begin{aligned} p_n(\mathbf{x}_n) &= p_{n-1}(\mathbf{x}_{n-1}) \cdot p(\mathbf{x}_n | \mathbf{x}_{n-1}) \\ &= p_{n-1}(\mathbf{x}_{n-1}) \cdot \prod_{i=1}^{|\mathbf{x}_n \setminus \mathbf{x}_{n-1}|} p(x_{n,i} | x_{n,1:(i-1)}, \mathbf{x}_{n-1}) \end{aligned}$$

where $x_{n,j:k}$ are the jth to kth random variables generated up to primitive n. The form of the per-variable conditional probability $p(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1})$ depends on the type of the variable $x_{n,i}$. If it is one of the procedural model's random choices, then the conditional probability is a function of the variable's parents in the program's dataflow graph and depends on the primitive distribution from which the variable is drawn (e.g. uniform, Gaussian):

$$p_r(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = p(x_{n,i}|\mathrm{par}(x_{n,i}))$$

If $x_{n,i}$ is an ordering choice, then the conditional probability is defined by an *ordering policy* π. This policy determines how to select the next subcomputation to continue when the currently-executing subcomputation finishes, or when a particle

synchronization barrier is reached (i.e. when a geometric primitive is generated).

We are concerned with two ordering policies. The first is the *deterministic* policy:

$$\pi_D(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = \begin{cases} 1 & \text{if } x_{n,i} = N(x_{n,1:(i-1)}, \mathbf{x}_{n-1}) \\ 0 & \text{otherwise} \end{cases}$$

where $N(x_{n,1:(i-1)}, \mathbf{x}_{n-1})$ is the number of subcomputations that could be continued at this point. This policy chooses the last option, equivalent to popping the top of a stack, with 100% probability. This behavior corresponds to running a program with depth-first execution ordering—normal SMC, in other words.

The second ordering policy of interest is the *stochastic* policy:

$$\pi_S(x_{n,i}|x_{n,1:(i-1)}, \mathbf{x}_{n-1}) = \frac{1}{N(x_{n,1:(i-1)}, \mathbf{x}_{n-1})}$$

which uniformly at random chooses a subcomputation to continue. This behavior corresponds to running a program with randomized execution order—the full SOSMC algorithm.

Thus far, we have only defined the *prior* distribution specified by the program itself. We already know how to sample from this distribution: run the program forward. Sampling only becomes challenging when we include a *likelihood* term that shapes the distribution. Our likelihood term is given by a user-provided score function $s(\cdot)$. Critically, this score function must be defined for partial execution traces $\mathbf{r}_n$, not just complete execution traces. The total, unnormalized posterior density at step n is

$$F_n(\mathbf{x}_n) = s(\mathbf{r}_n) \cdot p_n(\mathbf{x}_n)$$

The full, normalized probability distribution from which SOSMC samples at step n is

$$P_n^\pi(\mathbf{x}_n) = \frac{F_n(\mathbf{x}_n)}{Z_n^\pi} \tag{5.1}$$

where Z_n^π is the partition function which normalizes the distribution and depends on the ordering policy π. The final distribution in this sequence is P_N^π: the distribution

over complete runs of the program. As the number of SMC particles approaches infinity, their distribution approaches the target posterior density [100, 31]. Thus, SOSMC is an asymptotically-unbiased sampler for P_N^π.

In Appendix C, we show that the marginal distribution on generated models is the same under the stochastic ordering policy π_S as under the deterministic policy π_D. In other words, if we consider only the final state and not the order in which it was generated, then SOSMC draws samples from the desired distribution. The proof proceeds by marginalizing out the ordering choices $\mathbf{o}_N$ and showing that the two policies generate equivalent sets of complete traces $\mathbf{r}_N$. Finally, the N subscript indicates that our programs always terminate after a finite number of steps. The proof in Appendix C operates in this setting, but it also discusses programs that almost always terminate (i.e. terminate with probability one).

5.4 Implementation Using Stochastic Futures

To implement execution order randomization, we need a mechanism for interleaving the execution of different function calls with respect to one another. This requirement suggests looking at concurrent programming primitives. We settled on *futures*, a lightweight concurrency primitive that operates at the function call level [35]. Futures were originally designed for fine-grained parallelism, but we use them for a different interpretation of concurrency: sequential, interleaved programming. When called, a future may or may not begin executing, but it must finish executing when the program requests its return value. One common programming interface for futures allows for their creation by wrapping a function call with `future.create` and requesting their values by calling a `force` function. The interface to *stochastic* futures includes two more features:

- `future.switch()`: Switch control to and resume executing some other (random) active future. Our SOSMC implementation calls this function after every resampling step, allowing resampled particles to take different paths through the program as they advance.

- `future.finishall()`: Finish all active futures. Our programs generate geometry by appending to an implicit global model state, so most futures do not have a return value (e.g. the highlighted lines in Figure 5.2a). Our SOSMC implementation calls this function at the end of every program to force all such futures to finish.

To achieve the best performance with SOSMC, a procedural modeling program should use a stochastic future wherever it makes a branching decision predicated on a random choice. In our implementation, we insert these futures manually, since these situations are easy to identify in practice and typically occur near natural function call boundaries (e.g. `maybeGenWing` in Figure 5.2a). It should also be possible to automatically transform programs into this form using source-to-source compilation guided by simple static analysis (i.e. detecting when a random value flows into a conditional expression or statement).

Note that since function calls may execute in an arbitrary order, the program must be thread safe: any accesses to shared data can be reordered without changing program behavior. In our implementation, the only shared data structure is the global model state, and adding geometry to this state is an associative operation.

Implementing stochastic futures requires the ability to arbitrarily switch between different in-progress computations. Higher-level concurrency primitives are often implemented atop lower-level ones, such as threads. For stochastic futures, *coroutines* are a natural choice of implementation primitive. Coroutines are a generalization of subroutines that can suspend their execution, yield control to another coroutine, and then resume later. They were designed for sequential concurrency in the form of cooperative multitasking [17]. Algorithm 2 outlines an implementation of `switch`, `force`, and `finishall` in terms of asymmetric coroutines. Calling `finishall` initiates a loop that drives the random execution of futures, while `switch` and `force` determine when control returns to this loop.

We implement a prototype of SOSMC in Lua, with performance-critical components such as mesh voxelization and intersection implemented as high-performance extensions in Terra [18]. Following Algorithm 2, we implement stochastic futures using Lua's native coroutines. To perform weighted particle resampling, we use the

Algorithm 2 Implementing stochastic futures with coroutines

```
q ← { }                                          // A global queue of active futures
curr ← nil                                       // The currently-running future
procedure SWITCH()
    COYIELD()
procedure FORCE(future)
    // Suspend the forcing future until this future is finished
    q ← q \ {curr}
    future.waiters ← future.waiters ∪ {curr}
    return COYIELD()
procedure FINISHALL()
    // Randomly continue futures until all are finished
    while ¬ EMPTY(q) do
        f ← UNIFORMDRAW(q)
        CONTINUE(f)
procedure CONTINUE(future)
    curr ← future
    retvals ← CORESUME(future.co, future.args)
    future.args ← { }
    if COFINISHED(future.co) then
        // Reactivate any suspended futures that were waiting
        // for this one to finish
        for all w ∈ future.waiters do
            w.args ← retvals
            q ← q ∪ {w}
        q ← q \ {future}
```

well-known *systematic* resampling scheme for its simplicity and practical variance reduction properties; we also found *residual resampling* to work well [20]. The source code for our implementation can be found here: `https://github.com/dritchie/procmod`. For the comparisons to Metropolis Hastings in Section 5.5, we also implement Lightweight MH in Lua [119].

Both our prototype SMC and MH implementations must, on each iteration, replay the program trace from its beginning, though generated geometry and derived quantities are cached and not recomputed. This gives them both a quadratic time

complexity in the depth of the program. In Chapter 3, we demonstrated how to eliminate this overhead for MH. SMC for probabilistic programs can also be implemented without this overhead. Rather than replaying traces, particles could suspend and then resume at each sample/resample step. Resampling then requires copying suspended particles, which can be implemented efficiently with a construct like POSIX fork [79]. In a purely functional language, suspended particles can also be represented with continuations [30].

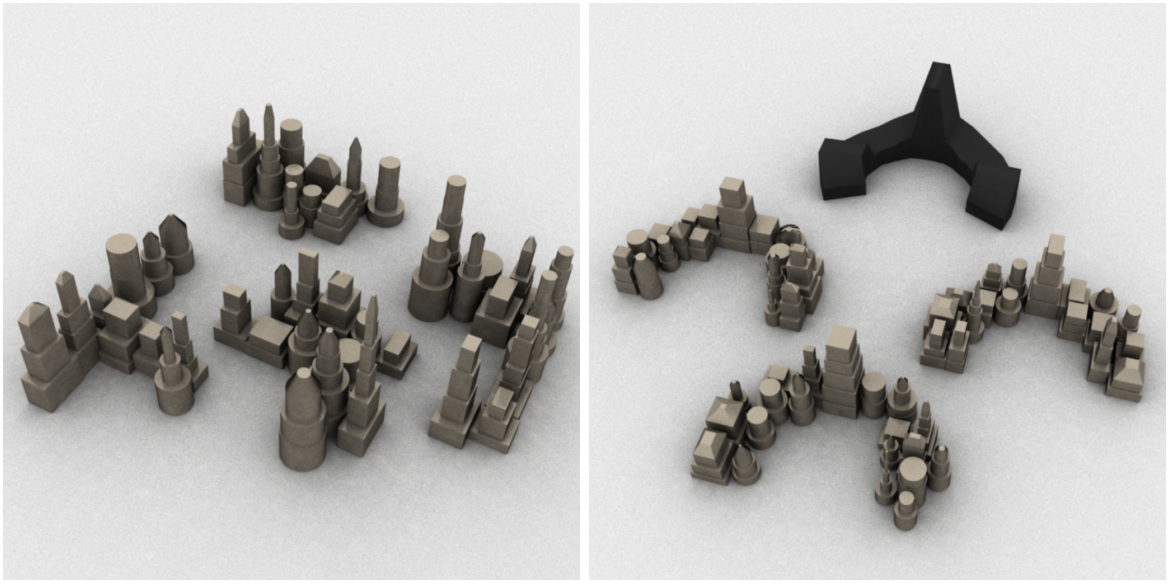
Finally, while our prototype implementation is serial, SMC is very straightforward to parallelize, which further enhances its performance potential. Particles can be evolved independently in parallel in the sampling phase and then gathered for the resampling phase using barrier synchronization.

5.5 Evaluation

We now demonstrate the ability of SOSMC to quickly and reliably generate high-quality procedural modeling samples. As test cases, we have chosen a variety of programs and controls that span a range of useful features, many of which have been explored previously in the literature [112]. We show that SOSMC can draw useful samples from these programs and controls, and that it generates higher-scoring samples than SMC or MH given small computational budgets. In all examples, we impose an additional score function which prevents geometry self-intersections by assigning a zero score to such configurations.

5.5.1 Volume Matching

It can be useful to control the overall 3D shape of a model via a rough geometric proxy. We implement this control volumetrically. If $V_{\mathbf{target}}$ is a target binary voxel grid defined over domain $\mathcal{D}$, and $V_{\mathbf{r}}$ is the voxelization of the model described by



Forward Sampling SOSMC Sampling

Figure 5.3: SOSMC sampling from a random building complex model with volume matching applied.

execution trace $\mathbf{r}$ onto $\mathcal{D}$, then the *volume matching* score function $s_{\mathbf{vmatch}}$ is

$$s_{\mathbf{vmatch}}(\mathbf{r}) = \mathcal{N}(\text{sim}(V_{\mathbf{r}}, V_{\mathbf{target}}), 1, \sigma) \cdot \mathcal{N}(\varepsilon_{\mathbf{out}}(\mathbf{r}), 0, \sigma)$$
$$\text{sim}(V_1, V_2) = \frac{1}{|\mathcal{D}|} \sum_{x \in \mathcal{D}} \mathbb{1}\{V_1(x) = V_2(x)\}$$

where $\text{sim}(V_1, V_2)$ returns a [0,1] similarity score for two voxel grids. $\varepsilon_{\mathbf{out}}(\mathbf{r})$ returns the maximum amount to which the model defined by $\mathbf{r}$ extends outside $\mathcal{D}$ along any dimension. The first normal term in $s_{\mathbf{vmatch}}(\mathbf{r})$ encourages similarity to the target volume. The second term penalizes growing beyond $\mathcal{D}$, where the target volume is not defined. We use a 2% error tolerance in all of our experiments ($\sigma = 0.02$) unless otherwise specified.

Figure 5.1 shows some examples of spaceships and trees sampled according to this score function using SOSMC. Figure 5.3 applies the same score function to encourage a building complex to take on a crescent-like shape.

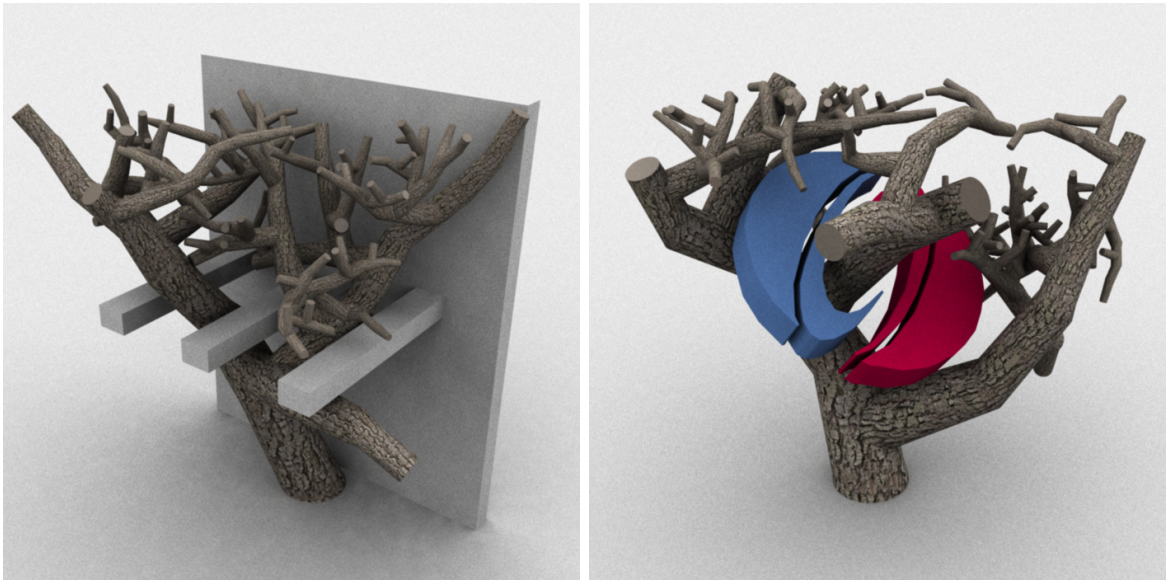


Figure 5.4: Using the object avoidance scoring function to make gnarly trees grow around obstacles.

5.5.2 Object Avoidance

Volume matching allows an artist to specify what regions of space a model should occupy; it can also be valuable to specify the space a model should *not* occupy. For this control, the user provides a set of objects with which the model should avoid contact. We rasterize these objects onto a binary voxel grid $V_{\mathbf{avoid}}$. The *object avoidance* score function $s_{\mathbf{avoid}}$ is then

$$s_{\mathbf{avoid}}(\mathbf{r}) = \prod_{x \in \mathcal{D}} \mathbb{1}\{V_{\mathbf{r}}(x) \uparrow V_{\mathbf{avoid}}(x)\}$$

where $\uparrow$ is logical NAND. This function imposes a hard constraint: it returns 0 if $V_{\mathbf{r}}$ and $V_{\mathbf{avoid}}$ have any mutually filled cells and 1 otherwise.

Figure 5.4 shows two examples of using object avoidance to generate trees that avoid obstacles. On the left, the tree avoids a wall with three protruding ledges; on the right, it grows through and around the SIGGRAPH logo. These examples also use a weaker version of the volume matching score function ($\sigma = 0.05$) to encourage

the trees to grow to a tall, full shape.

5.5.3 Image Matching

It is also useful to specify projective properties of a model, such as how it looks from a particular viewpoint or when lit from a particular angle. We implement this type of control through image-based comparisons. If $I_{\mathbf{target}}$ is a target binary image defined over domain $\mathcal{D}$, and $I_{\mathbf{r}}$ is a rendering of the model described by trace $\mathbf{r}$ onto $\mathcal{D}$, then the *image matching* score function $s_{\mathbf{imatch}}$ is

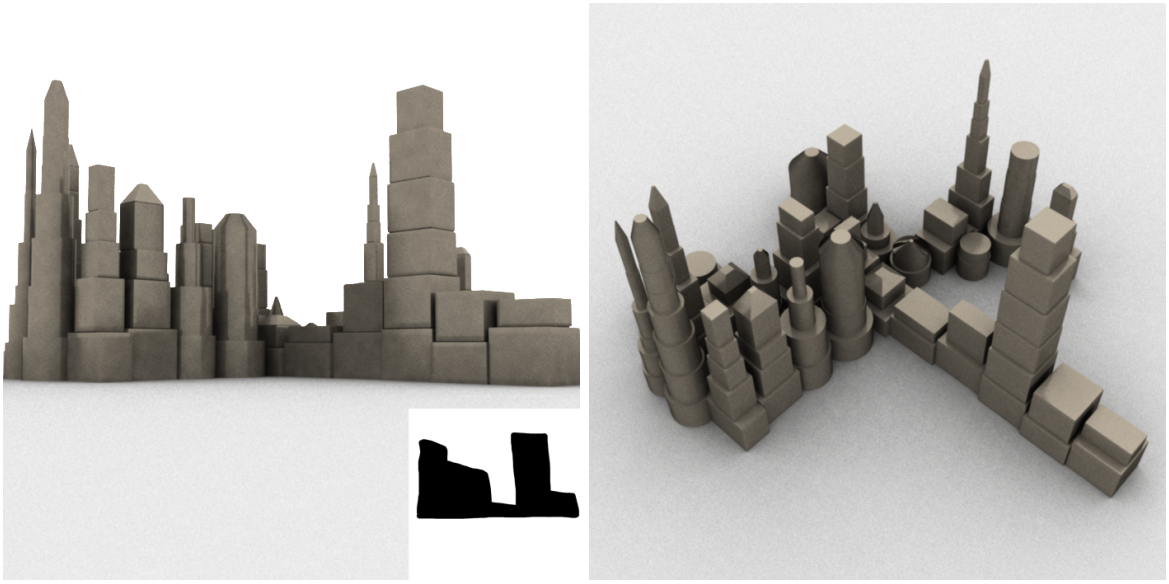
$$s_{\mathbf{imatch}}(\mathbf{r}) = \mathcal{N}(\mathrm{sim}(I_{\mathbf{r}}, I_{\mathbf{target}}), 1, \sigma)$$
$$\mathrm{sim}(I_1, I_2) = \frac{\sum_{x \in \mathcal{D}} W(x) \cdot \mathbb{1}\{I_1(x) = I_2(x)\}}{\sum_{x \in \mathcal{D}} W(x)}$$

where W is a 'weight image' defined over $\mathcal{D}$. The weight image allows users to draw strokes over parts of the image domain where strict matching is more or less important. For the results shown in this chapter, W is uniform unless explicitly shown. As with volume matching, σ is 0.02 unless otherwise specified.

Figure 5.5 shows a use of the image matching scoring function to enforce a target silhouette for a building complex when viewed from a particular angle. Note that the generated model is still free to exhibit random structure when viewed from other angles.

In Figure 5.6, we use image matching to control the profile of a spaceship. The generated models bear strong similarity to the target image when viewed from the front but are otherwise unconstrained, revealing diverse structure when viewed from other angles.

Figure 5.7 shows another use of image matching: controlling the shadows cast by toy blocks strewn about a floor. Here, we decrease the score error tolerance by an order of magnitude ($\sigma = 0.002$), use a weight image that places 10 times more weight on the outline of the target face image, and increase the particle count ($N = 1500$). These changes help SOSMC to match the fine details along the shadow silhouette edge. Again, the blocks in this example appear randomly arranged when viewed from



Front View Top View

Figure 5.5: The image matching scoring function is used to control the appearance of a building complex from a particular viewpoint. *(Left):* The model as viewed from the target viewpoint. *(Right):* The model viewed from above.

other angles.

In Figure 5.8, we use image matching to shape the shadow cast by a network of rectangular pipes. We lower the score error tolerance to $\sigma = 0.0005$ to encourage SOSMC to fill in the shadow as completely as possible while avoiding extrusions beyond the desired silhouette. For our implementation, this is more practical than increasing particle count due to the model's extreme depth.

5.5.4 Quantitative Evaluation

Table 5.1 shows timing statistics for the examples presented in this section. The second-to-last column shows the number of particles used by SOSMC; we find that 300 particles is sufficient to generate high-quality results in most cases. The last column reports the time taken to generate the example; for figures that show multiple output models, the time reported is the average time to generate them. All timing data was collected on an Intel Core i7-3840QM machine with 16GB RAM running

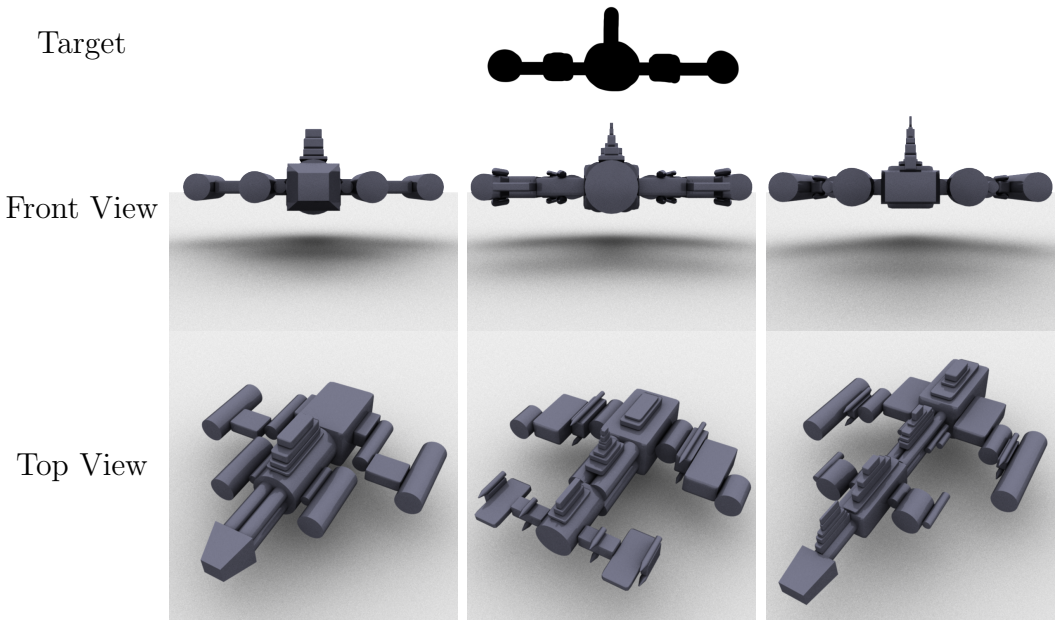


Figure 5.6: Using image matching to control the appearance of a spaceship's front profile. The SOSMC-sampled results closely match the target when viewed head on but exhibit a variety of structures when viewed from other angles.

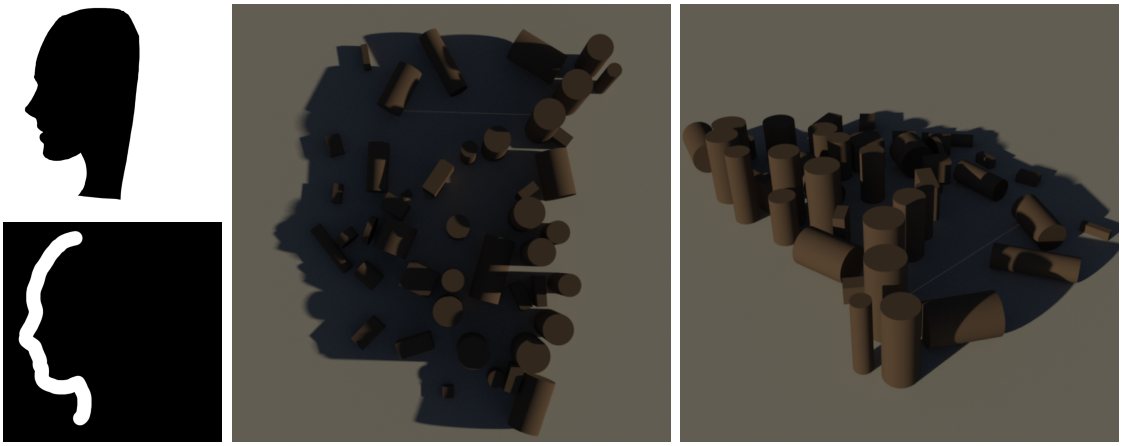


Figure 5.7: Using the image matching scoring function to control the shape of cast shadows in a scene with toy blocks scattered on a floor. Face silhouette image derived from a template created by Milliande Printables (`http://www.milliande-printables.com/face-silhouette-woman-stencil.html`).

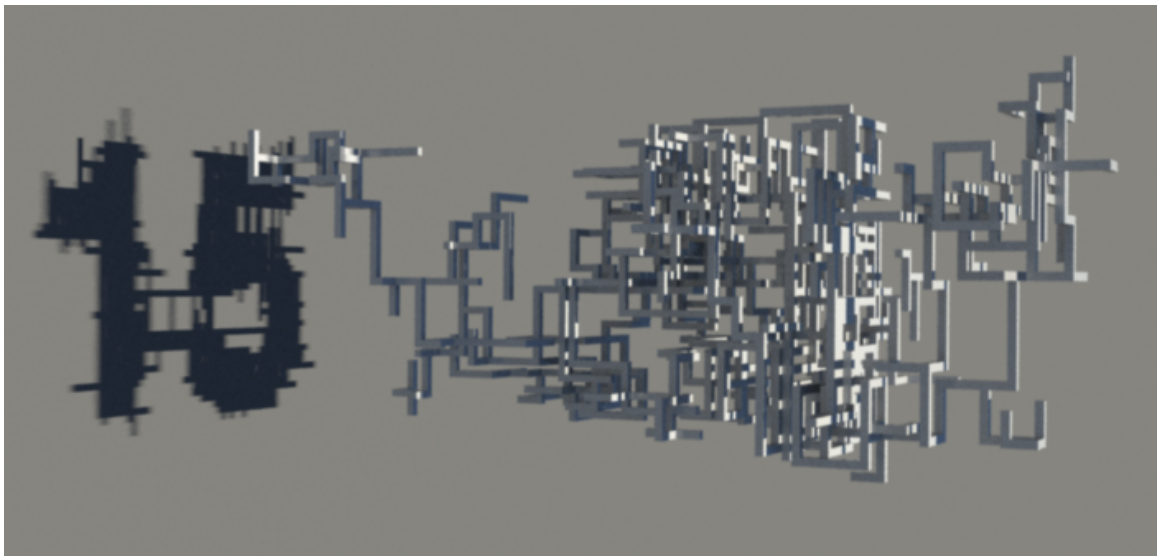


Figure 5.8: Using image matching to control the shadows cast by a network of pipes.

OSX 10.8.5. Times for simpler models, such as the spaceship, are already fast enough (a few seconds) to be used in interactive settings. As noted in Section 5.4, generation times for more complex models can be reduced by eliminating trace replay overhead or through parallelization.

We can also compare how well SOSMC, SMC, and MH generate high-scoring samples under different computational budgets. We are particularly interested in their behavior in low-budget scenarios. As test cases, we use the spaceship, building complex, and tree programs under volume constraints. These programs all exhibit recursive structure, but of a different nature: the spaceship program spawns recursive paths (wings, etc.) from a single recursive spine (the body), whereas the building complex and tree programs generate components in a multiply-branching, tree-recursive style.

We find that MH requires additional tuning to achieve peak performance. Specifically, it generates better results with a score function tempered down to a 0.5% error tolerance ($\sigma = 0.005$). We also experimented with parallel tempering but found it not to perform better than normal MH when run for the same amount of time on sequential hardware. If run on parallel hardware—as in e.g. Talton et al. [112]—it

Program	*Control*	*Figure*	N	*Time* (s)
Spaceship	$s_{\mathbf{vmatch}}$	5.1	300	3.09
Gnarly Trees	$s_{\mathbf{vmatch}}$	5.1	300	598.34
Building Complex	$s_{\mathbf{vmatch}}$	5.3	300	24.14
Gnarly Trees	$s_{\mathbf{avoid}} \cdot s_{\mathbf{vmatch}}$	5.4	100	164.25
Building Complex	$s_{\mathbf{imatch}}$	5.5	300	38.44
Spaceship	$s_{\mathbf{imatch}}$	5.6	300	7.33
Toy Blocks	$s_{\mathbf{imatch}}$	5.7	1500	135.91
Pipes	$s_{\mathbf{imatch}}$	5.8	100	675.81

Table 5.1: Timing data for all procedural modeling examples shown in this chapter. N is the number of particles used by SOSMC.

could perform better, but for fair comparison, we would also have to run SMC in parallel. SMC could then process more particles in the same amount of time, improving its performance as well.

In our comparison experiment, we run SMC and SOSMC for particle counts ranging from 10 to 1000. At each particle count, we also run MH, giving it as much time as an average SOSMC run takes to complete at that particle count. We run each algorithm 10 times, take the highest score for each run, and record the mean and variance of those high scores. Figure 5.9 shows the results of this experiment. On the left, we plot mean highest score against increasing computational budget; line thickness is proportional to variance in highest score. Our implementation computes all quantities in log-probability space, so the scores shown are log scores.

For the spaceship example, SOSMC starts with higher scores than either SMC or MH, which both require a significant amount of computation to reach the same score level. SOSMC also achieves consistently low variance in scores (evidenced by the thin orange lines in Figure 5.9, left), suggesting that it reliably generates high-scoring results on every run. SMC suffers from the order-sensitivity problems discussed in Section 5.2 but appears to overcome them when given enough particles. MH fares

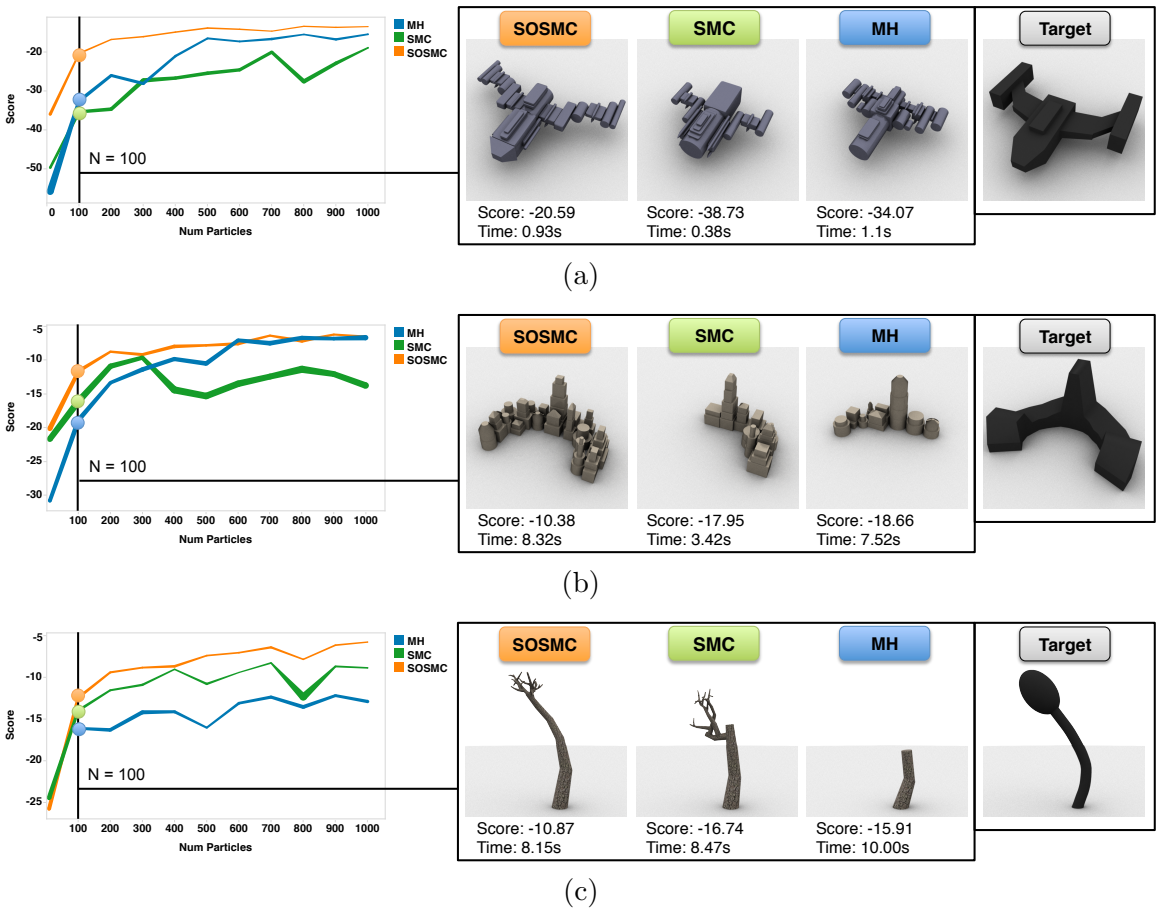


Figure 5.9: A comparison of SOSMC, SMC, and MH in generating high-scoring outputs with limited computation time. *(Left)* Maximum score achieved by each method, averaged over 10 runs, as computational budget increases. Line thickness is proportional to variance in high scores over those runs. SMC and SOSMC use the same number of particles; MH runs for as long as SOSMC takes to run on average. *(Right)* Representative samples generated by each method given a computational budget of 100 particles (or equivalent average running time, for MH). SOSMC consistently outperforms both SMC and MH in reliably generating high-quality samples at small budgets.

slightly better than SMC in terms of score, and its outputs are also qualitatively more diverse, featuring more interesting variations.

In the building complex example, SMC again suffers from order sensitivity. While it can generate good results, it often fails to generate both sides of the target crescent

curve (Figure 5.9b, right), leading to high variance in scores (Figure 5.9b, left). MH fares better than SMC and does eventually match SOSMC's scores at high budgets. At low budgets, however, SOSMC generates good volume matches, whereas MH does not have enough time to reliably do so (Figure 5.9b, right). MH requires twice as much computation as SOSMC to consistently score above -10, the threshold above which results appear consistently 'good' for this example.

For the tree example, SMC's performance is close to SOSMC's, since the target shape has linear structure with branching only at the end. However, order-sensitivity is still an issue, as SMC sometimes generates models that use a large branch where continuing the trunk would be more natural (Figure 5.9c, right). MH also performs well overall on this example, but there is a persistent gap between its performance and that of SOSMC. MH's proposals—which randomly re-generate subtrees—can fail to discover the long structure of the target shape, especially at low budgets (Figure 5.9c, right).

5.6 Chapter Summary

This chapter introduced SMC to the task of controlled procedural modeling. We developed the SOSMC algorithm and the stochastic future to handle the multiple possible sequentializations of a procedural modeling program. We demonstrated SOSMC's ability to generate high-quality results for a variety of programs and controls, and we showed that it reliably generates better results under small computational budgets than both depth-first SMC and MH.

5.6.1 Limitations

SOSMC will not always succeed for all possible programs and score functions. SMC is known to be susceptible to 'garden paths,' or execution traces that look promising for much of their runtime only to become undesirable later on [58]. In settings where such paths exist, SOSMC could conceivably perform worse than depth-first SMC, as it may randomly discover garden paths that depth-first SMC cannot follow. For

such problems, the ability to revise past decisions is critical, so MCMC or hybrid SMC/MCMC approaches work better [1].

SMC also needs random choices to be interleaved with evidence (i.e. geometry generation) to work well. If too many random choices are made up-front, the program 'overcommits' itself and proceeds like simple forward sampling. Fortunately, most hierarchical, recursive procedural models can be written in interleaved style. Simple data flow analysis could be used to push random choices as close as possible to their dependent geometry, if the program is not already written in this way.

In addition, SMC can suffer from the 'sample impoverishment' problem: repeated resampling tends to kill off all but one or a few particle execution histories, resulting in a final set of particles whose early execution histories are identical. For procedural modeling programs, this behavior manifests in many near-duplicates in the final set of sampled output models. Ideally, SMC would deliver as many unique samples as it has particles, and there exist a variety of impoverishment-fighting techniques that could help realize this goal, though at the cost of more computation time [28, 63]. MCMC algorithms suffer from a similar problem in the form of 'mode lock,' wherein the MCMC chain becomes stuck in a small, localized region of the state space.

5.6.2 Scalability

The examples presented in this chapter are relatively simple, using from dozens up to a few hundred primitives, but we believe that SOSMC should scale well to models of increasing complexity. In terms of depth complexity (i.e. how many primitives the program generates), an implementation that avoids trace replay, such as a continuation-based implementation, should be able to maintain nearly-constant work per SMC timestep. Some scoring functions, such as intersection testing, could still become more expensive as depth complexity increases, however.

In terms of breadth complexity (i.e. the program's approximate branching factor), a high branching factor results in more possible execution orderings, which could require more particles to explore. The results presented in this chapter suggest that SOSMC can work well up to branching factor 4 (the Building Complex program) with

a reasonable number of particles; future work could further explore this question.

Chapter 6

Learning to Sample using Neural Guides

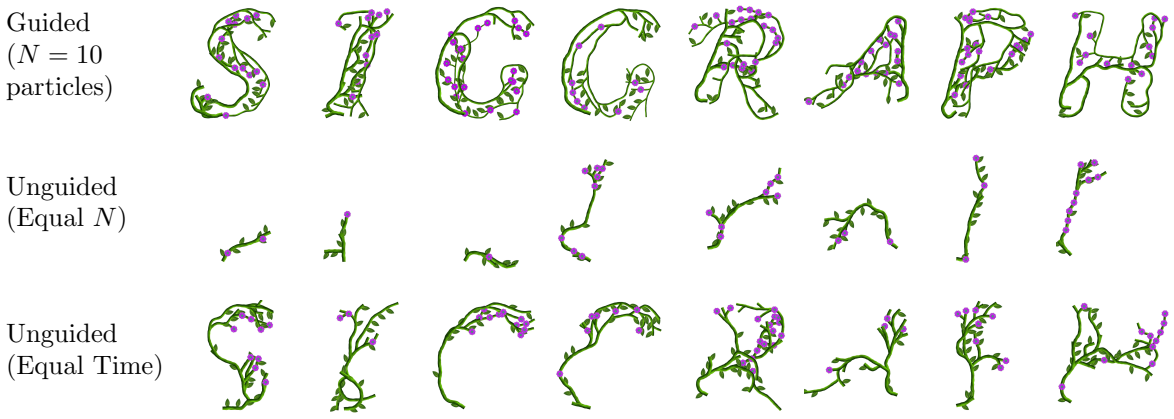


Figure 6.1: *(Top Row)* Used as an importance sampler for Sequential Monte Carlo with $N = 10$ particles, our neurally-guided procedural models generate shape-matching results for each of the above letters in about a second. *(Middle Row)* The naïve, unguided procedural model does not converge to recognizable results using the same number of particles ($N = 10$). *(Bottom Row)* The unguided model does better, but still does not reliably converge, when given the same amount of computation time as the guided model ($\approx 1\,\text{sec}$).

Even with the advances described in the previous three chapters, generating high-quality results from constrained procedural modeling programs still requires considerable computation time: thousands of iterations for MH, or hundreds of particles for SMC. This limits their usability for interactive applications.

Fundamentally, sampling from constrained procedural models is challenging because the constraints implicitly define complex (often non-local) dependencies not present in the unconstrained procedural model (i.e. the prior). Can we instead make these dependencies *explicit* by encoding them in the models' generative logic? Such an explicit model could simply be run forward to generate constraint-satisfying results.

In this chapter, we propose a method for automatically learning an approximation to such a perfect explicit model. Our method leverages advances in deep learning: it augments the procedural model with neural networks that control how the model makes random choices, based on what partial output the model has generated thus far. We call such a model a *neurally-guided procedural model*. The neural networks are expressive enough to capture many implicit dependencies induced by the constraints.

Building on our work from the previous chapter, we train neurally-guided procedural models using constraint-satisfying example outputs generated via SMC. Once trained, these models can be used as intelligent SMC important samplers. Our approach thus enables 'bootstrapping' samplers which train on their own outputs and become more efficient over time. Or, the system can invest time up-front generating and training on many examples, effectively 'pre-compiling' an efficient sampler.

We demonstrate our method through experiments with L-system-like procedural models with image-based soft constraints (Figure 6.1). For a given constraint satisfaction score threshold, our neurally-guided procedural model can generate results which reliably achieve that threshold using 10-20x fewer particles and up to 10x less compute time than an unguided procedural model.

We give a high-level overview of our approach in Section 6.2 and then present the mathematical foundations of our method in Section 6.3. In Section 6.4, we describe how to implement neurally-guided procedural models with image-matching constraints. Finally, we evaluate the performance of those models in Section 6.5.

6.1 Related Work

Probabilistic Inference for Procedural Modeling As surveyed in Section 2.1 and discussed in Section 5.1, many research projects have used Bayesian probabilistic inference to control procedural models: constraining the shape of a 3D object [112, 91], creating functionally-plausible and aesthetically-pleasing furniture arrangements [69, 129], coloring in patterns [62], and dressing virtual characters [131] are a few recent applications. Our work aims to make such systems more efficient: neurally-guided procedural models can capture many of the dependencies introduced by constraint likelihood functions, so samplers need fewer samples to find good results.

In recent work similar in spirit to our own, Dang and colleagues built a system which modifies a procedural grammar so that its output distribution reflects user preference scores given to example outputs [16]. Like us, they seek a model whose generative logic captures dependencies induced by a likelihood function (in their case, a Gaussian process regression over user-provided examples). Their method works by splitting non-terminal symbols in the original grammar, giving it more degrees of freedom to capture more dependencies. This approach works well for discrete dependencies, such as ensuring all floors of a building have the same architectural style. In contrast, our method captures dependencies using neural networks, making it better suited for complex, continuous constraint functions, such as shape-fitting.

Guided Procedural Modeling The non-probabilistic approaches to controlling procedural models mentioned in Section 5.1 are again relevant in this chapter. The seminal work on open/environmentally-sensitive L-systems developed a formalism by which L-systems could query their spatial position and orientation [86, 75]. This ability allows them to prune their growth to an implicit surface. Recent follow-up work extends this technique to larger models by decomposing them into separate guide regions with limited interaction [4]. These guide methods were carefully designed for the specific problem of fitting procedural models to shapes. In contrast, our method *learns* how to guide procedural models and is generally applicable to constraints which can be expressed as a likelihood function.

Neural Networks for Procedural Modeling Previous work has found other ways to apply neural networks to procedural modeling. One recent project uses neural networks as computationally inexpensive proxies for costly scoring functions in an inverse urban procedural modeling setting [114]. Another uses an autoencoder network to learn a low-dimensional representation space in which it is easy to explore the variability in a procedural model's output [132]. Our use of neural networks differs from both of the above projects, as we use them to capture constraint-induced dependencies via feedforward functions.

Neural Variational Inference Our method is also inspired by recent work in variational inference [72, 88, 53]. These algorithms use neural networks to define more expressive parametric families of probability distributions. They train stochastic deep belief networks and autoencoders, primarily modeling distributions over images for computer vision applications. Our method uses a different learning objective, and we focus on training procedural models with more complex recursive control flow.

The Neural Adaptive Sequential Monte Carlo algorithm is most similar to our method; it uses a similar learning objective and aims to train more efficient SMC importance samplers [33]. However, they focus on inference in time series models, such as nonlinear state space models.

6.2 Approach

In this section, we motivate and outline the process of creating, training, and using neurally-guided procedural models.

6.2.1 Motivation

We motivation our approach using a simple program `chain` that recursively generates a random sequence of linear segments, constrained to match a target image. Figure 6.2a shows the text of this program, along with samples generated from it (drawn in black) against several target images (drawn in gray). Chains generated by running

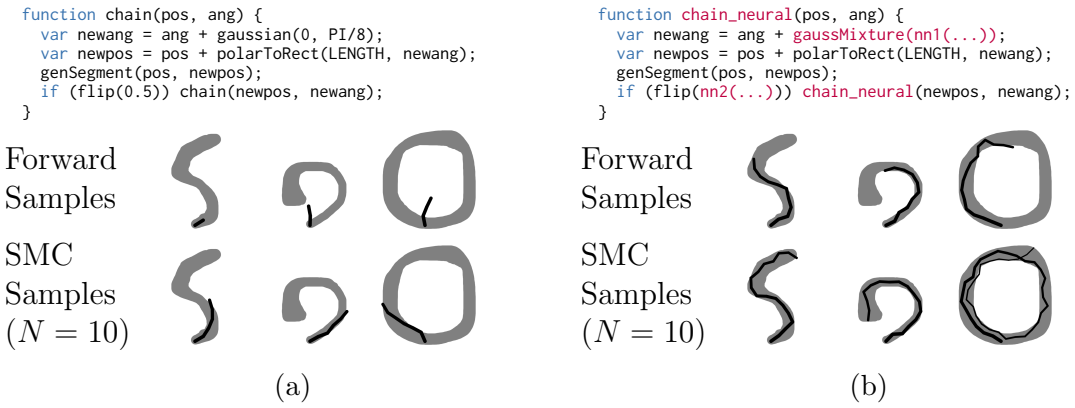


Figure 6.2: Transforming a simple linear chain model into a neurally-guided procedural model. *(a)* The original program. When the program's output (shown in black) is constrained to match a target image (shown in gray), forward sampling gives poor results. SMC sampling performs better but requires far more than 10 particles to achieve good results for all targets. *(b)* The neurally-guided program, where parameters of random choices are computed via neural networks. The neural nets receive the target image and all previous random choices as input (abstracted as "..."; see Figure 6.3b). Once trained, forward sampling from this program adheres closely to the target image, and SMC with 10 particles consistently produces good results.

the program forward do not match the targets, since forward sampling is oblivious to the constraint. Instead, we can generate constrained samples using Sequential Monte Carlo (SMC) [91]. SMC generates multiple samples, or *particles*, in parallel, resampling them at each step of the program to favor constraint-satisfying partial outputs. This results in final chains that more closely match the target images. However, the algorithm requires many particles—and therefore significant computation—to produce acceptable results. Figure 6.2a shows that $N = 10$ particles is not sufficient.

In an ideal world, we would not need costly inference algorithms to generate constraint-satisfying results. Instead, we would have access to an 'oracle' program, `chain_perfect`, that perfectly fills in the target image when run forward. What form might this program take? At each step, it would need access to the target image, to know where to grow the chain next. It would also need to see the output it has already generated, to know when it has filled the target and can stop growing the chain.

Our insight is that while oracle programs such as `chain_perfect` can be difficult or impossible to write by hand, it is possible to *learn* a program `chain_neural` that comes close. Figure 6.2b shows our approach. For each random choice in the program text (e.g. `gaussian`, `flip`), we replace the parameters of that choice with the output of a neural network. This neural network's inputs (abstracted as "...") include the target image as well the choices the program has made thus far. The network thus shapes the distribution over possible choices, guiding the programs's future output based on the target image and its past output. These neural nets affect both continuous choices (e.g. angles) as well as control flow decisions (e.g. recursion): they dictate where the chain goes next, as well as whether it keeps going at all. For continuous choices such as `gaussian`, we also modify the program to sample from a mixture distribution. This helps the program handle situations where the constraints permit multiple distinct choices (e.g. in which direction to start the chain for the circle-shaped target image in Figure 6.2).

When properly trained, a neurally-guided procedural model such as `chain_neural` generates constraint-satisfying results more efficiently than its un-guided counterpart. Figure 6.2b shows example outputs from `chain_neural`. Forward samples adhere closely to the target images, and SMC with 10 particles is sufficient to produce chains that fully fill the target shape. The next sections of the chapter describe the process of building and training these neurally-guided procedural models in more detail.

6.2.2 System Overview

Figure 6.3 shows a high-level overview of our workflow for defining, training, and using neurally-guided procedural models. It consists of the following steps:

Transform The procedural model is first transformed by inserting one neural network for each random choice in the program text and turning continuous random choices into mixture distributions (Figure 6.3a-b). The network receives as input the constraint (e.g. a target image) and all previously-made random choices (shown grayed out in Figure 6.3a-b) and outputs the parameters for the choice (e.g. Gaussian means, variances, and mixture weights). We perform this transformation manually;

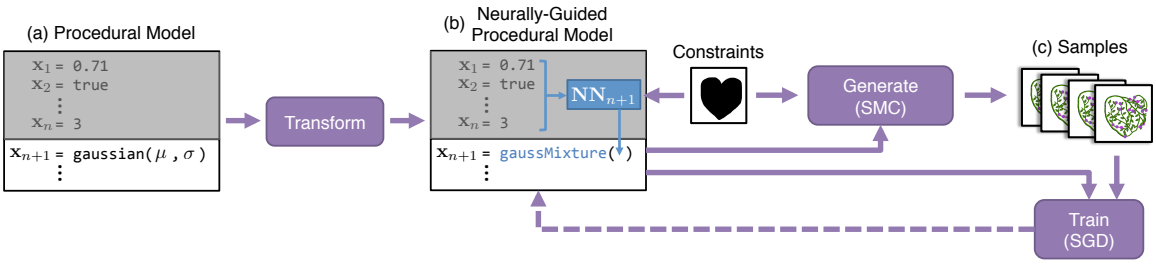


Figure 6.3: Overview of our approach. *(a)* We start with a procedural model: a program that makes a sequence of random choices $\mathbf{x}_1 \ldots \mathbf{x}_m$. *(b)* The procedural model is transformed into a neurally-guided procedural model by adding a neural network at each random choice. The network predicts the parameters of the random choice as a function of the constraints and the previous random choices (shown grayed-out). *(c)* An SMC sampling algorithm generates samples from the constrained procedural model. A stochastic gradient learning algorithm then trains the neurally-guided procedural model to maximize the probability of generating these samples.

it could be automated via source-to-source compilation. The neural networks can capture multiple different constraints, but an appropriate architecture for them depends on the generative paradigm and the output domain of the procedural model (e.g. images, 3D models, etc.) In Section 6.4, we present an architecture for 2D L-system-like procedural models which generate images. In particular, we describe how our implementation converts the previous random choices into a fixed-width vector appropriate for input to a neural net.

Generate Given a constraint, such as a target image, Sequential Monte Carlo generates samples from the constrained procedural model (Figure 6.3c). Our system uses the version of SMC presented earlier in Chapter 5, where particles are resampled after the program generates a new piece of geometry. It also uses the trained models as importance samplers for this SMC algorithm when generating final results.

Train The generated samples are then used to train the neural networks: the desired outcome is a set of network parameters that make the model more likely to generate these samples when run forward. We derive the learning objective in Section 6.3 and the details of our stochastic gradient learning method in Section 6.4. The trained

neurally-guided model can then quickly generate more samples, which can serve as further training data for refining the model, if desired.

6.3 Mathematical Foundations

Having outlined our approach, we now formally define neurally-guided procedural models. For our purposes, a *procedural model* is a generative probabilistic model of the following form:

$$P_{\mathbf{M}}(\mathbf{x}) = \prod_{i=1}^{|\mathbf{x}|} p_i(\mathbf{x}_i; \Phi_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}))$$

Here, $\mathbf{x}$ is the vector of random choices the model makes as it executes (the dimensionality of $\mathbf{x}$ may be variable, as with recursive procedural models such as stochastic L-systems). The p_i's are local probability distributions from which each successive random choice is drawn. p_i is parameterized by a set of parameters (e.g. mean and variance, for a Gaussian distribution), which are determined by some function Φ_i of the previous random choices $\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}$. The total probability density is the product of these local probabilities, according to the chain rule.

A *constrained procedural model* is a procedural model whose probability distribution is modulated by some likelihood function $\ell(\mathbf{x}, \mathbf{c})$, i.e. a scoring function indicating how well an output of the model satisfies some constraint $\mathbf{c}$. For example, $\mathbf{c}$ could be an image, with $\ell(\cdot, \mathbf{c})$ measuring similarity to that image. By Bayes' rule:

$$P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c}) = \frac{1}{Z} \cdot P_{\mathbf{M}}(\mathbf{x}) \cdot \ell(\mathbf{x}, \mathbf{c})$$

where Z is a normalizing constant. The set of all constraints $\mathbf{c}$ supported by the procedural model forms the constraint space $\mathcal{C}$ (e.g. all images, all binary mask images, etc.)

A *neurally-guided procedural model* modifies a procedural model by, for each local

probability p_i, replacing the parameter function Φ_i with a neural network:

$$P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta) = \prod_{i=1}^{|\mathbf{x}|} \tilde{p}_i(\mathbf{x}_i; \mathrm{NN}_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{c}; \theta))$$

The neural nets receive the previous random choice values and the constraint as input, and are themselves parameterized by θ. $\tilde{p}_i$ is a mixture distribution if random choice i is continuous; otherwise, $\tilde{p}_i = p_i$.

In training a neurally-guided procedural model, our goal is to find the parameters θ such that $P_{\mathbf{GM}}$ is as close as possible to $P_{\mathbf{CM}}$ for all supported constraints. Formally, we seek to minimize the conditional KL divergence $D_{\mathrm{KL}}(P_{\mathbf{CM}}||P_{\mathbf{GM}})$. Given some prior distribution $P(\mathbf{c})$ over constraints $\mathbf{c} \in \mathcal{C}$, our optimization objective is:

$$\begin{aligned}
&\min_\theta D_{\mathrm{KL}}(P_{\mathbf{CM}}||P_{\mathbf{GM}}) \qquad (6.1)\\
&\quad = \min_\theta \mathbb{E}_{P(\mathbf{c})}\left[\mathbb{E}_{P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})}\left[\log \frac{P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})}{P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta)}\right]\right]\\
&\quad = \min_\theta \mathbb{E}_{P(\mathbf{c})}\left[\mathbb{E}_{P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})}\left[\log P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c}) - \log P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta)\right]\right]\\
&\quad = \max_\theta \mathbb{E}_{P(\mathbf{c})}\left[\mathbb{E}_{P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})}\left[\log P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta) - \log P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})\right]\right]\\
&\quad = \max_\theta \mathbb{E}_{P(\mathbf{c})}\left[\mathbb{E}_{P_{\mathbf{CM}}(\mathbf{x}|\mathbf{c})}\left[\log P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta)\right]\right]\\
&\quad \approx \max_\theta \frac{1}{N}\sum_{s=1}^{N} \log P_{\mathbf{GM}}(\mathbf{x}_s|\mathbf{c}_s;\theta)\\
&\qquad \mathbf{x}_s \sim P_{\mathbf{CM}}(\mathbf{x})\,,\ \mathbf{c}_s \sim P(\mathbf{c})
\end{aligned}$$

In the last step, we approximate the expectations with an average over a finite set of samples $\mathbf{x}_s, \mathbf{c}_s$ drawn from the constrained procedural model $P_{\mathbf{CM}}$ using SMC and the constraint prior $P(\mathbf{c})$. If we view these samples as a training data set, then this optimization objective is simply maximizing the log-likelihood of the training data under the neurally-guided model $P_{\mathbf{GM}}$.

With such a training set in hand, optimization proceeds via stochastic gradient

ascent using the gradient

$$\begin{aligned} &\nabla \log P_{\mathbf{GM}}(\mathbf{x}|\mathbf{c};\theta) \\ &\quad = \sum_{i=1}^{|\mathbf{x}|} \nabla \log \tilde{p}_i(\mathbf{x}_i; \mathrm{NN}_i(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \mathbf{c}; \theta)) \end{aligned} \tag{6.2}$$

Is is worth noting that $D_{\mathrm{KL}}(P_{\mathbf{CM}}||P_{\mathbf{GM}})$ is not the only measure of distance between probability distributions we could have used. In particular, several related works have used the other direction of KL divergence, $D_{\mathrm{KL}}(P_{\mathbf{GM}}||P_{\mathbf{CM}})$, due to its attractive properties: it requires training samples from $P_{\mathbf{GM}}$, which are much less expensive to generate than samples from $P_{\mathbf{CM}}$. It is the optimization objective used in many variational inference algorithms [120, 45, 72] as well the REINFORCE algorithm for reinforcement learning [117]. When used for procedural modeling, however, this objective leads to models whose outputs lack diversity, making them unsuitable for generating visually-varied content. This behavior is due to a well-known property of the objective: minimizing it produces approximating distributions that are overly-compact, i.e. concentrating their probability mass in a smaller volume of the state space than the true distribution being approximated [64].

6.4 Implementation

In this section, we describe an implementation of neurally-guided *accumulative* procedural models: models that iteratively or recursively add new geometry to a structure. Most growth models, such as L-systems, are accumulative [84]. This is in contrast with other modeling paradigms: spatial subdivision, such as architectural split grammars [74]; object subdivision, such as fractal terrain [59]; or simulation, such as erosion-based terrain [115]. For our purposes, a procedural model is accumulative if, while executing, it provides a 'current position': a point $\mathbf{p}$ from which geometry generation will continue. We focus on 2D models ($\mathbf{p} \in \mathbb{R}^2$), though the techniques we present extend naturally to 3D.

We first describe the neural network architecture used by the neurally-guided

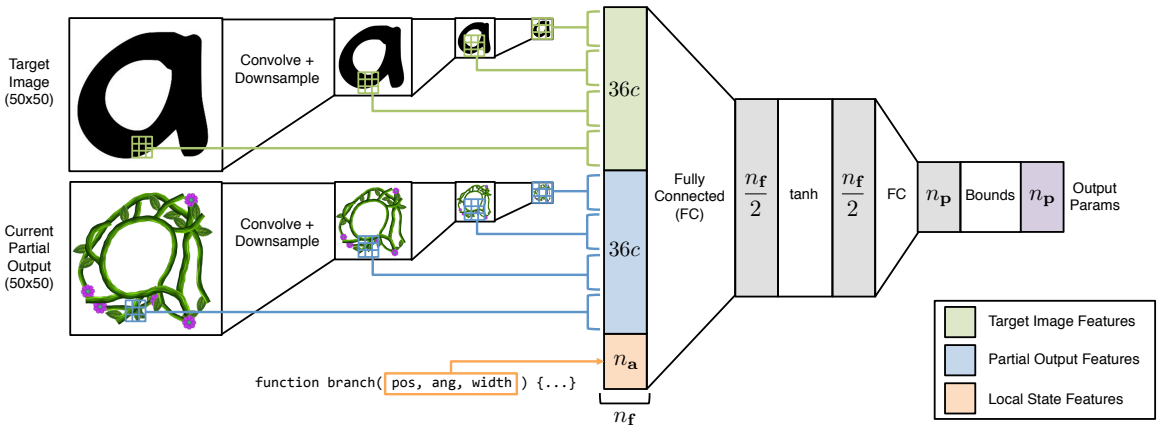


Figure 6.4: Neural network architecture for image-matching procedural models. The network uses a multilayer perceptron which takes a vector of features as input and outputs the parameters for a random choice probability distribution. The input features come from three sources. *Local State Features* are the arguments to the function in which the random choice occurs. *Target Image Features* come from 3x3 pixel windows of the target image, extracted at multiple resolutions, around the procedural model's current position. *Partial Output Features* are analogous windows extracted from the partial image the model has generated. All of these features can be computed from the target image and the sequence of random choices made thus far.

models before giving details on how we train them.

6.4.1 Network Architecture

Our neural networks take as input the constraint $\mathbf{c}$ (in this case, a target image) and all previously-made random choices, and output the parameters of a random choice. Figure 6.4 shows our network architecture. We use a multilayer perceptron (MLP) architecture, because it is simple, easy to scale, and is a universal function approximator [97, 15]. Our MLP takes $n_\mathbf{f}$ inputs, has one hidden layer of size $n_\mathbf{f}/2$ with a tanh nonlinearity, and has $n_\mathbf{p}$ outputs, where $n_\mathbf{p}$ is the number of parameters the random choice expects. Since some parameters are bounded (e.g. Gaussian variance must be positive), each output is remapped via an appropriate bounding transform (e.g. e^x for non-negative parameters). We experimented with more hidden layers but found that this did not improve performance.

The inputs for the MLP come from several sources, each providing the network with decision-critical information. All of these features can be computed from the target image and the choices-so-far; for efficiency, we compute them incrementally as the program runs.

Local State Features The first set of relevant data is the local state of the procedural model: its current position $\mathbf{p}$, the current orientation of any local reference frame, its current recursion depth, etc. Our networks access this information via the arguments of the function in which the random choice occurs. We extract all $n_{\mathbf{a}}$ scalar arguments, normalize them to $[-1, 1]$, and pass them to the MLP.

Target Image Features To make appropriate decisions for matching a target image, the network must have access to that image. The raw pixels provide too much data; we need to summarize the relevant image contents. Convolutional neural networks reduce an image to a fixed-width feature vector but are aimed at classification tasks: they detect features but are intentionally invariant to where those features occur [55].

Instead, we use a different, location-sensitive architecture. We extract a 3x3 window of pixels around the model's current position $\mathbf{p}$. We do this at four different resolution levels, with each level computed by convolving the previous level with a 3x3 kernel and then downsampling via a 2x2 box filter. For a image with channel depth c, this results in $36c$ features. Together, these features summarize what the target image looks like from the procedural model's current position, where resolution decreases with distance. This architecture is similar to the foveated 'glimpses' used in recent work on neural models of visual attention [73].

Partial Output Features The target image features provide the network with information it needs to generate matching content with *accuracy* (e.g. how to stay within a target shape) However, they do not provide the information necessary to achieve *completeness* (e.g. how to completely fill in a target shape). To give the network this capability, we also extract multi-resolution windows from the partial

output image generated by the procedural model thus far. This adds another $36c$ input features.

The parameters θ of this architecture consist of the weights and biases for both fully-connected layers in the MLP, as well as the kernel weights and biases for the three convolution + downsampling layers on each image. Each network typically has around several thousand such parameters. For example, given a program with four local features (position x, position y, angle, width) which targets a one-channel image, the network that predicts the parameters of a four-component Gaussian mixture has 3466 parameters.

6.4.2 Training

We train neurally-guided procedural models by stochastic gradient ascent using the gradient in Equation 6.2. Our system computes this gradient via backpropagation from the $\log \tilde{p}_i$'s to the neural network parameters θ. We use the Adam algorithm for stochastic gradient optimization, with $\alpha = \beta = 0.75$ and an initial learning rate of 0.01 [52]. We found that a mini-batch size of one worked best in our experiments: more frequent gradient updates led to faster convergence than less-frequent-but-less-noisy updates. We terminate training after 20000 gradient updates.

6.4.3 Implementation Details

We implemented our prototype system in the Javascript-based probabilistic programming language WebPPL [30], with neural networks implemented using an open-source Javascript library for neural computation.[1] The source code for our system is available at `https://github.com/dritchie/webppl/tree/variational-neural-gl`.

6.5 Experiments

In this section, we qualitatively and quantitatively evaluate how well our neurally-guided procedural models capture image-based constraints. We first describe our

[1] https://github.com/dritchie/adnn

Figure 6.5: Example images from our datasets.

databases of target images before presenting the details of several experiments. All timing data reported in this section was collected on an Intel Core i7-3840QM machine with 16GB RAM running OSX 10.10.5.

6.5.1 Image Datasets

As shown in Equation 6.1, each training sample from a procedural model must be paired with a constraint $\mathbf{c}$ drawn from a prior $P(\mathbf{c})$ over possible constraints. During training, we sample target images uniformly at random from a database of training images. In our experiments, we use the following image collections:

- **Scribbles**: A set of 49 binary mask images drawn by hand with the brush tool in Photoshop. These were designed to cover a range of possible shapes with thick and thin regions, high and low curvature, and different self-intersections.
- **Glyphs**: A subset of 197 glyphs from the FF Tartine Script Bold typeface. Consists of all glyphs which have only one foreground connected component

and at least 500 foreground pixels when rendered at 129x97.

- **PhyloPic**: A set of 35 images from PhyloPic, a database of silhouettes for plants, animals, and other organisms.[2]

When using these images for training, we augment the datasets by also including a horizontally-mirrored duplicate of each image. We also annotate each image with a starting point and starting direction from which to initialize the execution of a procedural model. Figure 6.5 shows some representative images from each collection.

6.5.2 Shape Matching

In our first set of experiments, we train neurally-guided procedural models to capture 2D shape matching constraints, where the target shape is specified as a binary mask image. If $\mathcal{D}$ is the spatial domain of the image, and $I(\mathbf{x})$ is the function which renders the current partial output defined by random choices $\mathbf{x}$, then the likelihood function for this constraint is

$$\ell_{\mathbf{shape}}(\mathbf{x}, \mathbf{c}) = \mathcal{N}(\frac{\text{sim}(I(\mathbf{x}), \mathbf{c}) - \text{sim}(\mathbf{0}, \mathbf{c})}{1 - \text{sim}(\mathbf{0}, \mathbf{c})}, 1, \sigma_{\mathbf{shape}}) \tag{6.3}$$

$$\text{sim}(I_1, I_2) = \frac{\sum_{\mathbf{p}\in\mathcal{D}} w(\mathbf{p}) \cdot \mathbb{1}\{I_1(\mathbf{p}) = I_2(\mathbf{p})\}}{\sum_{\mathbf{p}\in\mathcal{D}} w(\mathbf{p})}$$

$$w(\mathbf{p}) = \begin{cases} 1 & \text{if } I_2(\mathbf{p}) = 0 \\ 1 & \text{if } ||\nabla I_2(\mathbf{p})|| = 1 \\ w_{\mathbf{filled}} & \text{if } ||\nabla I_2(\mathbf{p})|| = 0 \end{cases}$$

where $\mathcal{N}$ is the normal distribution. This function encourages the rendered image to be similar to the target mask, where similarity is normalized against the target's similarity to an empty image $\mathbf{0}$. Each pixel $\mathbf{p}$'s contribution to the similarity is weighted by $w(\mathbf{p})$, determined by whether the target mask is empty, filled, or has an edge at that pixel. We use $w_{\mathbf{filled}} = 0.\bar{6}$, so that empty and edge pixels are worth 1.5 times as much as filled pixels. This encourages the program to match

[2] http://phylopic.org

perceptually-important contours before filling in flat regions. We set $\sigma_{\mathbf{shape}} = 0.02$ in all experiments.

We wrote a WebPPL program which recursively generates vines with leaves and flowers and then trained a neurally-guided version of this program to capture the above likelihood. The model was trained on 10000 sample traces, each generated using SMC with 600 particles. Target images were drawn uniformly at random from the Scribbles dataset. Each sample took on average 17 seconds to generate; parallelized across four CPU cores, the entire set of samples took approximately 12 hours to generate (later in this section, we show that far fewer samples are actually needed). Training took 55 minutes in our single-threaded Javascript implementation. These times could be reduced with more efficient implementations (e.g. leveraging GPUs for training).

Figure 6.1 shows example outputs from the vines program. The weighting scheme of $\ell_{\mathbf{shape}}$ causes the geometry to adhere to target shape contours, making the shape recognizable without cluttering interior regions and obscuring the vines' structural characteristics. This behavior is not easy to achieve with a purely generative space-filling approach such as environmentally-sensitive L-systems [86], but it is simple to specify with constraints. The top row outputs were generated using 10-particle SMC with the trained neurally-guided model, which reliably produces recognizable results. In contrast, 10-particle SMC with the unguided model produces totally unrecognizable results (middle row). Because the neural networks make the guided model more computationally-expensive to evaluate, a more equitable comparison is to give the unguided model the same amount of wall-clock time as the guided model—this correponds to $\sim$ 50 particles, in this case (bottom row). While the resulting outputs fare better, the target shape is still obscured. We should also note that the unguided model is unpredictable at such low particle counts; results of even this quality took many tries to obtain at 50 particles. In practice, we find that the unguided model needs $\sim$200 particles to reliably match the performance of the guided model. Figure 6.6 shows more outputs from the vines program, and Figure 6.7 shows example outputs from a neurally-guided procedural lightning program.

Figure 6.8 shows a quantitative comparison between five different models on the

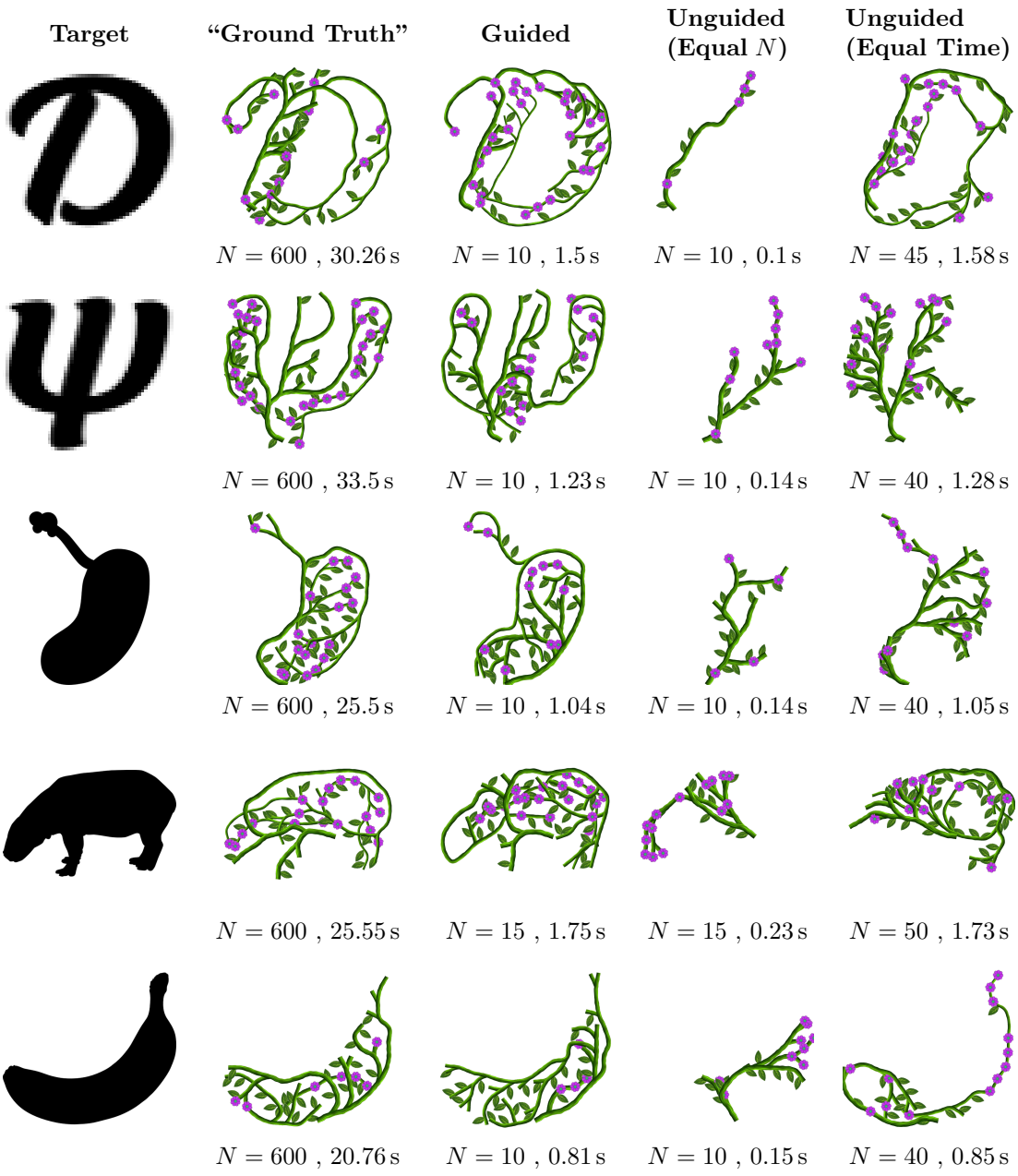
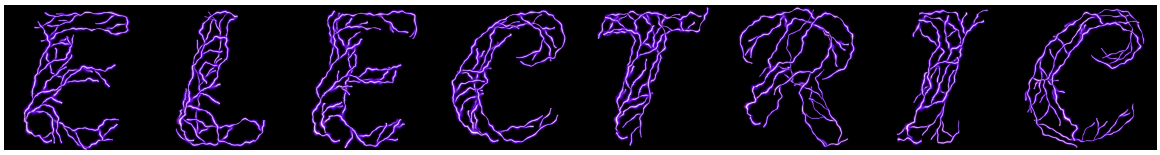


Figure 6.6: Using Sequential Monte Carlo to make a vine-growth procedural model match target images. N is the number of SMC particles used. The *"Ground Truth"* column shows an example result after running SMC with the unguided model with a large number of particles ($N = 600$); these images represent the best possible result for a given target. Our neurally-guided procedural models can generate results of nearly this quality in a couple seconds; the unguided model struggles given the same number of particles or the same computation time.



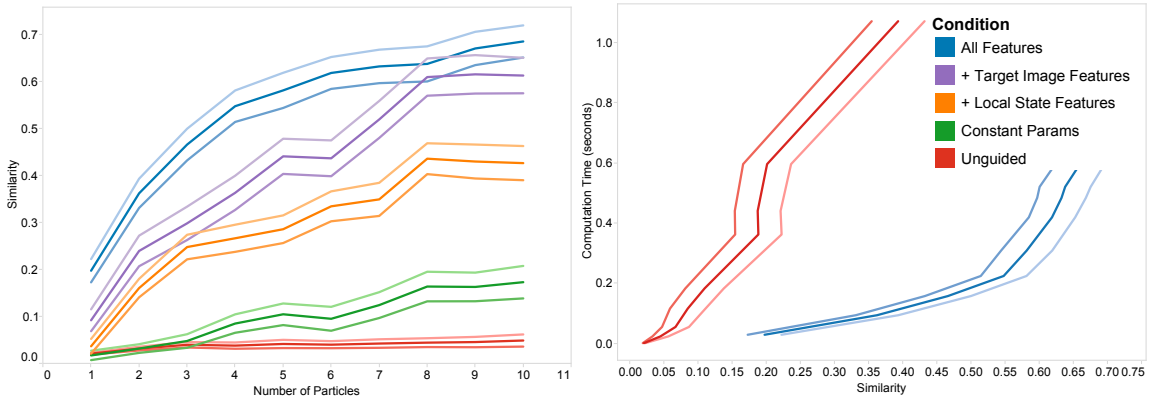
0.99 s 0.81 s 1.01 s 1.03 s 0.9 s 1.16 s 0.86 s 1.08 s

Figure 6.7: Targeting letter shapes with a neurally-guided procedural lightning program. Generated using SMC with 10 particles; compute time required is shown below each letter. Best viewed on a high-resolution display.

shape matching task:

- **Unguided**: The original, unguided procedural model.
- **Constant Params**: The neural network for each random choice is simply a set of constant parameters; training this model finds the optimal constants. This is also known as a *partial mean field approximation* [120].
- **+ Local State Features**: Adding the local state features described in Section 6.4.
- **+ Target Image Features**: Adding the target image features described in Section 6.4.
- **All Features**: The full neural net architecture described in Section 6.4, including local state features, target image features, and partial output features.

We test each model on the images in the Glyph dataset and report the median normalized similarity-to-target achieved (i.e. argument one to the Gaussian in Equation 6.3). Figure 6.8a plots this average similarity as the number of SMC particles increases. The performance of the neurally-guided models improves with the addition of more features; at 10 particles, the full model is already near the peak performance asymptote. Figure 6.8b shows the wall-clock time each method requires as the desired average similarity increases. The vertical gap between the two curves shows the speedup given by neural guidance, which can be as high as 10x. Note that we trained on the Scribbles dataset but tested on the Glyphs dataset; these results suggest that our models can generalize to qualitatively-different previously-unseen images.



(a) (b)

Figure 6.8: Performance comparison for the shape matching problem. *"Similarity"* is median normalized similarity to target mask, averaged over all targets in a test dataset. Lines drawn in lighter shades show 95% confidence bounds. *(a)* Performance as the number of SMC particles increases. The neurally-guided model achieves higher average similarity as more features are added. *(b)* Computation time required as desired similarity increases. The vertical gap between the two curves indicates speedup. Despite the neurally-guided model being more expensive to evaluate, it still reliably generates high-similarity results significantly faster than the unguided model.

Figure 6.9 shows the benefit of using mixture distributions for continuous random choices in the guided model. The experimental setup is the same as in Figure 6.8. We compare a model which uses 4-component mixture distributions with a no-mixture model. The with-mixtures model provides a noticeable performance boost, which we alluded to in Section 6.2: when matching complex shapes with junctions and intersections, such as the crossing of the letter 't,' the program benefits from modeling uncertainty at these points with multi-modal distributions. We found 4 mixture components sufficient for our examples.

We also investigate how the number of training samples affects performance. Figure 6.10 plots the median similarity at 10 particles as training set size increases. Performance increases rapidly for the first few hundred samples before appearing to level off (the noise in the curve is due to randomness in neural net training initialization). This suggests that ∼1000 sample traces is sufficient, which may seem surprising, as many published deep learning systems require millions of training examples [55]. In

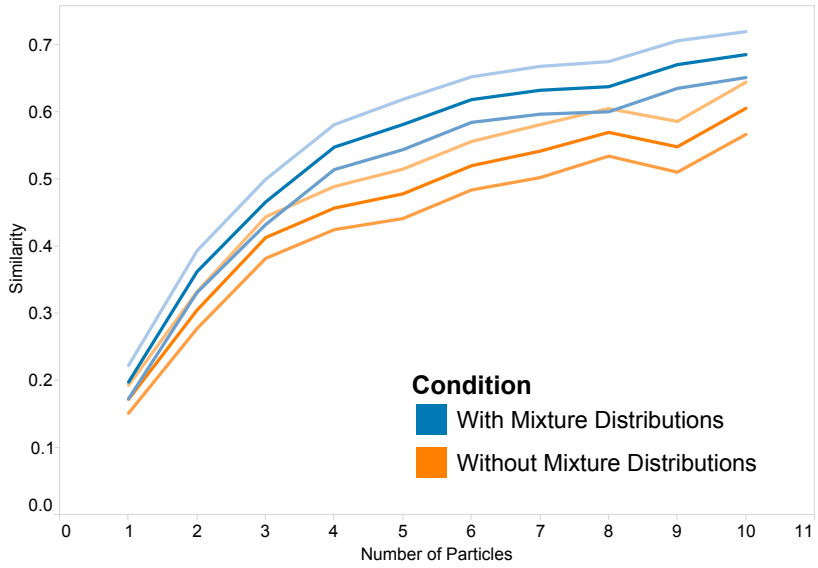


Figure 6.9: The effect of guiding continuous random choices with mixture distributions. Using 4-component mixtures for all continuous random choices provides a noticeable boost in performance.

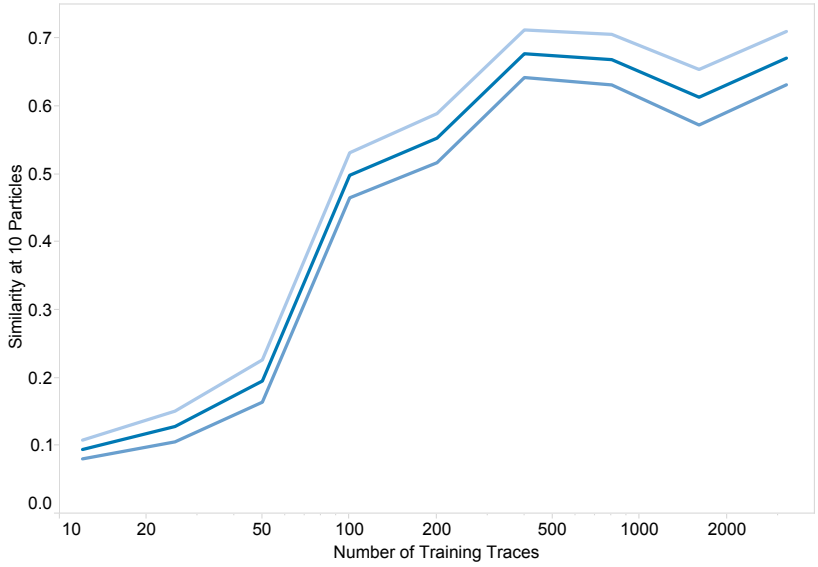


Figure 6.10: The effect of training set size on performance (at 10 SMC particles), plotted on a logarithmic scale. Average similarity-to-target increases sharply for the first few hundred sample training traces, then appears to plateau at around 1000 traces. Noise in the plot is due to randomness in neural net training, as different training sessions converge to different local optima of the learning objective function.

our case, each training trace contains up to thousands of random choices, each of which provides a learning signal—in this way, the training data is "bigger" than it appears. Our implementation can generate 1000 samples in just over an hour using four CPU cores. As mentioned previously, this time could be reduced by 'boostrapping' the system: training on smaller subsets of data and using the partially-learned model to generate further data faster.

6.5.3 Stylized "Circuit" Design

Thus far, we have focused on image-matching constraints. However, the architecture we have presented can learn other types of image-based constraints. In this section, we constrain the vines program to generate outputs which resemble stylized circuit designs.

Dense packing of long wire traces is one of the most striking visual characteristics of circuit boards. To achieve dense packing, we encourage the program to fill a certain percentage τ of the image ($\tau = 0.5$ in the subsequent results). To mimic the appearance of traces, we encourage the output image to have a dense, high-magnitude gradient field, as the vines program can best achieve this result by creating many long rectilinear or diagonal edges. These constraints result in the following likelihood:

$$\begin{aligned} \ell_{\mathbf{circ}}(\mathbf{x}) &= \mathcal{N}(\text{edge}(I(\mathbf{x})) \cdot (1 - \eta(\text{fill}(I(\mathbf{x})), \tau)), 1, \sigma_{\mathbf{circ}}) \\ \text{edge}(I) &= \frac{1}{|\mathcal{D}|} \sum_{\mathbf{p} \in \mathcal{D}} ||\nabla I(\mathbf{p})|| \\ \text{fill}(I) &= \frac{1}{|\mathcal{D}|} \sum_{\mathbf{p} \in \mathcal{D}} I(\mathbf{p}) \end{aligned} \tag{6.4}$$

where $\eta(x, \bar{x})$ is the relative error of x from $\bar{x}$ and $\sigma_{\mathbf{circ}} = 0.01$. Finally, we also include a separate term that penalizes the program from generating geometry outside the bounds of the image; this encourages the program to fill in a rectangular "die"-like region.

To guide this program, we use the same architecture as before, minus the target image features (since there is no target image). We train the neurally-guided

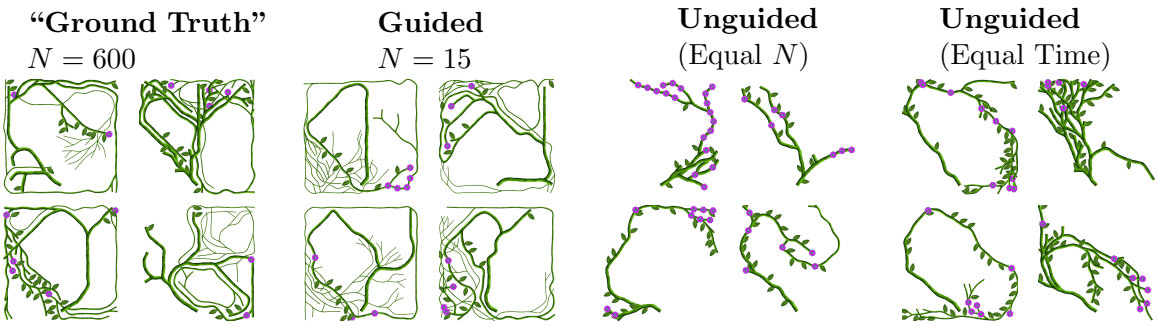
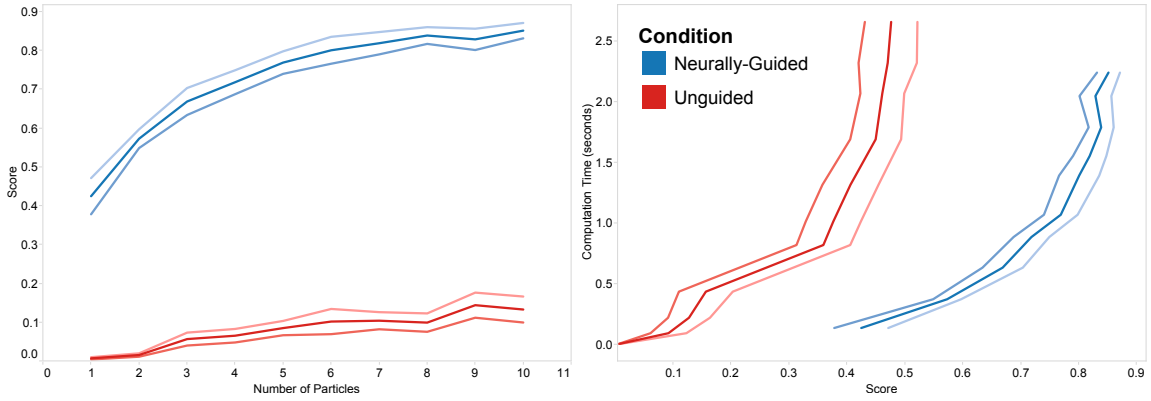


Figure 6.11: Constraining the vine-growth program to generate circuit-like patterns. The *"Ground Truth"* outputs took around 70 seconds to generate; the outputs from the guided model took around 3.5 seconds.



(a) (b)

Figure 6.12: Performance comparison for the circuit design problem. *"Score"* is median normalized score (i.e. argument one to the Gaussian in Equation 6.4), averaged over 50 runs. The neurally-guided version achieves significantly higher average scores than the unguided version given the same number of particles or the same amount of compute time.

model using 2000 traces generated using SMC with 600 particles. Sample generation took about 10 hours on four CPU cores, and training took just under two hours. Figure 6.11 shows some outputs from this program, and Figure 6.12 shows a performance comparison between unguided and neurally-guided models for this task. As with the shape matching examples, the neurally-guided model generates high-scoring results significantly faster than the unguided model.

6.6 Chapter Summary

This chapter introduced neurally-guided procedural models: constrained procedural models that use neural networks to capture constraint-induced dependencies. We developed a mathematical framework for defining and training such models. We also described a specific neural architecture for accumulative models that generate images. Finally, we evaluated the performance of neurally-guided models, demonstrating that they can generate high-quality results significantly faster than unguided models.

One limitation of our system is its need for training data, which must be generated via expensive inference. This can be a significant up-front cost, especially for computationally-expensive models. Thus, our method is not well-suited for scenarios where the procedural model changes rapidly, such as speeding up the inner loop of a development and debugging cycle. Instead, it is best suited for scenarios where the model is fixed, such as deploying a finalized procedural model as part of a design tool. It may be particularly attractive for online, multi-user deployments, where the system can gather example results from the community, periodically retrain, and push the updated procedural model to users.

Our method is also not well-suited for capturing hard constraints, which some visual effects necessitate (e.g. symmetries), as it requires a continuous probability for each training sample. While hard constraints can sometimes be usefully approximated with tight soft constraints, neural networks such as ours are best at approximating noisy and/or random functions, not precise, deterministic relationships. Other techniques are needed for generatively capturing these kinds of constraints.

Chapter 7

Conclusions and Future Directions

In this thesis, we have advocated the use of probabilistic programming languages (PPLs) for procedural modeling and design. We showed examples of how complex models can be expressed in relatively clear, concise code, requiring users to provide domain knowledge but not requiring them to be inference experts. We then pointed out that such model complexity leads to inference challenges: wasted computation, tight constraints, branching structures, and the expense of random search.

To address these problems, we introduced new inference techniques to the field of procedural modeling and design. To eliminate wasted computation during MH proposals, we developed the C3 system for lightweight, incrementalized MCMC, leading to order-of-magnitude speedups. To efficiently explore design spaces shaped by tight constraints, we demonstrated how Hamiltonian Monte Carlo (HMC) could sample from previously-intractable programs such as stable stacking structures. To handle branching, we developed Stochastically-Ordered Sequential Monte Carlo, showing that it can more quickly and reliably find high-probability outputs for procedural shape-matching problems. Finally, to make the random search methods of inference less random, we introduced neurally-guided procedural models which can learn how to satisfy constraints up to an order of magnitude faster.

7.1 Interoperability

The four methods we presented in this thesis do not each have to operate in a vacuum; they can be assembled into an ecosystem of interoperable inference components.

This is the approach we have taken in the ongoing development of the WebPPL probabilistic programming language [30]. In our implementation, C3 and HMC are MCMC *kernel* functions, which can be programmatically composed to create new kernels—for example, one that proposes to discrete random choices using C3 and to continuous choices using HMC. In addition, WebPPL's Sequential Monte Carlo (SMC) implementation supports rejuvenation, i.e. changing past choices in the sequence using MCMC [28]. Any MCMC kernel can be used for rejuvenation, including C3 and HMC.

Neurally-guided procedural models also interoperate with the other inference algorithms we have discussed. In Chapter 6, we used SMC to generate training data for neural guides, but one can in fact use any inference algorithm to generate this data, including MH. And while we focused on using the trained neural guides as importance distributions for SMC, they can also be used as proposal distributions for MH. We have begun efforts to integrate these concepts into WebPPL for general programs.

7.2 Future Work

To move PPLs out of the domain of research and into real-world applications, there is still much to be done. Inference must be be even faster, approaching interactive rates, to be usable in real-time settings such as games and interactive design tools. And the easier it is to write probabilistic programs, the faster and wider they can spread. Along these lines, opportunities for future work fall into several broad themes:

7.2.1 Program Analysis & Transformation

Production PPL systems will need to squeeze every last drop of exploitable structure out of programs, analyzing and then manipulating them into their most efficient

forms.

Non-standard interpretations have already proven useful in this space [118]. The automatic differentiation (AD) transformation used by HMC is one such example: the program is transformed to compute derivatives in addition to values. As mentioned in Section 4.5, smooth interpretation can take this idea further by automatically constructing a differentiable approximation to any program (even those with loops and conditionals). These ideas have been applied to SMC as well. Abstract particle algorithms automatically construct and operate on a hierarchy of simplified variable domains, allowing inference to converge at a coarse resolution before tackling fine-scale details [109, 107]. We expect to see more innovation along these lines in years to come.

These types of transformations can introduce computational overhead which careful program analysis can remove. In C3, dependency analysis could help determine statically which random choices flow to which other functions, allowing the system to perform fewer of the runtime input equality checks introduced by the caching transform. C3 also only requires continuations at random choice points, yet the standard CPS transform applies to the entire program. Detecting and fusing blocks of purely deterministic code before applying the CPS transform (or collapsing them post-transform) could improve performance. A similar phenomenon occurs with HMC, as standard AD implementations generate one computation graph node for each primitive math operation. Programs could make simpler graphs by statically "lowering" all operations that happen within a single basic block [9]. Or, already-constructed graphs could be simplified by run-time tracing followed by symbolic simplification [126, 19].

Neural guidance is also a form of program transformation. Specifically, it replaces the dataflow paths that flow into random choices with new dataflow paths made out of neural networks. More extensive transformations are also possible, such as changing the control flow of the program. Transformations like this might make it possible for guide programs to capture constraints that dataflow-replacement alone cannot.

7.2.2 Runtime Systems

Production PPL systems will also need to run programs with the fastest possible platforms and execution environments. Many of the early research PPLs in existence today are dynamically-typed and run in interpreters [29, 65] or virtual machines with JIT compilers [121, 30], which limits their possible performance. One noteworthy exception is Quicksand, a language we implemented in earlier work to experiment with statically-typed, compiled probabilistic programs [89]. We found some performance improvements, though the lower-level nature of the language makes some inference algorithms difficult to implement with peak asymptotic efficiency (e.g. MH and SMC, since neither first-class continuations nor first-class functions are available in the host language, Terra). Another notable exception is Probabilistic C, which uses OS-level multi-processing to run SMC and Particle MCMC methods on multiple cores [79]. Leveraging parallel hardware like this will be especially important for future practical PPL systems.

Along these lines, GPUs still remain largely untapped for probabilistic programming. There are wide-open opportunities to exploit GPU parallelism either for inference algorithms themselves (e.g. parallel execution of SMC particles) or for the constituent parts of a probabilistic program (e.g. accelerating matrix computations in complex likelihood functions).

7.2.3 Amortized Inference

As we showed in Chapter 6, PPL systems can see significant performance boosts by learning from their own outputs, either over time or as a batch pre-process.

Our work on neurally-guided procedural models is a first step in this direction, but there are many more opportunities along these lines. We would like to extend the idea of neural guidance to work with more general programs, not just L-system-like programs. Recent work on recurrent visual attention models is particularly promising, as these networks could help a guide program learn both what to look at in its partial output, as well as what choices to make given what it has seen [73, 25].

Amortized inference is a broader concept that has implications beyond neural

guide programs. For example, in our experiments with using SMC to control procedural model shapes, we use the current partial likelihood score as the particle resampling weight. While this can work, it also results in greedy behavior, since the weights do not take into account the expected future likelihood score of each particle. It might be possible to learn a model of this expected future score, in a similar fashion to how the AlphaGo system learns a value function for Markov Decision Processes [99]. And in SOSMC, we have thus far only used a uniform stochastic policy for future selection—it might be possible to learn non-uniform stochastic policies over time. Such policies would gradually develop an 'intuition' for which execution branches are likely to be fruitful given the program's execution history.

7.2.4 Authoring & Editing

No matter how fast PPL inference becomes, the programs being run through inference have to come from somewhere. Thus far, we have assumed that a programmer writes the program in full. But to make PPL inference accessible and desirable to the widest range of creative people, it will need to accommodate those who are not comfortable with extensive programming.

One seductive idea is to induce probabilistic programs from a (ideally small) collection of example outputs. Recent research has had some limited successes in this area, by restricting the class of supported programs and abstractions over programs [111, 42]. Such general program induction is a highly underconstrained and challenging problem, but has large potential impact if solved.

Putting a human in the loop of the induction process could make the problem easier, as well as make the results more satisfying to the user since she can exert direct control over them. On approach to such "partial program induction" is bidirectional editing interfaces, where a user can either edit code to affect its output, or edit outputs to affect the code. Recent research has had some limited successes along these lines, inferring constant parameters for deterministic programs that generate 2D vector graphics [13]. Further work is needed to extend these concepts to probabilistic programs, 3D output domains, and more structural program edits. This is a promising

avenue for making PPLs more accessible and understandable, so more creative people can express themselves with these tools.

Appendix A

C3 Speedup Experiment Details

All models use synthetic data. To obtain the throughput values plotted, each model was run for 1000 MH iterations.

The HMM model uses 10 discrete latent states and 10 discrete observable states. The Model Size parameter maps to the length of the observed sequence as: Length of observed sequence $= 100 \cdot$ Model Size. Thus, observed sequence length ranges from 100 to 1000.

The LDA model uses 10 topics, a vocabulary of 100 words, and 20 words per document. The Model Size parameter maps to the number of observed documents as: Num observed documents $= 5 \cdot$Model Size. Thus, num observed documents ranges from 5 to 50, and total number of observed words ranges from 100 to 1000.

The GMM model uses 4 components. The Model Size parameter maps to the number of observed data points as: Num observed data points $= 100 \cdot$ Model Size. Thus, the number of observed data points ranges from 100 to 1000.

The HLR model uses scalar data sequences of length 5. The Model Size parameter maps to the number of data sequences as: Num data sequences $= 20 \cdot$ Model Size. Thus, the number of data sequences ranges from 20 to 200, and the total number of scalar data points ranges from 100 to 1000.

Appendix B

HMC Model Specifications

B.1 Color Compatibility Model

Our color compatibility model uses soft constraints simplified from the model by Lin and colleagues [62]. We include saturation, adjacent lightness difference, and adjacent perceptual difference constraints, since these factors had high learned weights in the original model. The perceptual difference constraints (implemented as distance in CIELAB color space) help distinguish image regions without excessive hue contrast. The lightness difference constraints help prevent equiluminant adjacent regions which can cause perceived "vibrations" and unstable-looking shapes. Constraints are parameterized differently if they are applied to foreground (FG) or background (BG) regions:

Saturation (BG): $\texttt{softeq}(\frac{\sqrt{a^2+b^2}}{\sqrt{a^2+b^2+L^2}}, 0.3, 1.0)$

Saturation (FG): $\texttt{softeq}(\frac{\sqrt{a^2+b^2}}{\sqrt{a^2+b^2+L^2}}, 0.7, 1.0)$

Lightness Diff (FG-BG): $\texttt{softeq}(\frac{|\Delta L|}{100}, 0.3, 0.4)$

Lightness Diff (FG-FG): $\texttt{softeq}(\frac{|\Delta L|}{100}, 0.2, 0.4)$

Perceptual Diff: $\texttt{softeq}(\frac{\sqrt{\Delta L^2+\Delta a^2+\Delta b^2}}{300}, 0.3, 0.2)$

where L, a, b are the color coordinates of a region in CIELAB color space, and ΔL, Δa, Δb are differences between the coordinates of adjacent regions. 100 is the maximum lightness value, and 300 is the maximum CIELAB distance. The rough shape of these constraints are based on those in the original model. Our model is a weighted sum of these factors for each region and each pair of adjacent regions. Saturation constraints are weighted by the region area, while pairwise adjacent constraints are weighted uniformly. As in the Lin et al. model, we represent and perform inference on color random variables in RGB space to ensure that all generated results use colors that can be visualized.

We also add constraints to enforce semantic properties where needed. We consider two types of additional constraints in our experiments:

Same-Chroma $\texttt{softeq}(\frac{\sqrt{\Delta a^2+\Delta b^2}}{282.9}, 0, \frac{\sigma}{282.9})$

Lightness-Relation $\texttt{softeq}(\frac{\Delta L}{100}, 0.15, \frac{\sigma}{100})$

where 282.9 is the maximum chroma difference. We set σ to 5 in our experiments. The Same-Chroma constraint dictates that two colors should have the same chromatic content (i.e. the same color irrespective of lightness). The Lightness-Relation constraint enforces a precise directional separation between the lightnesses of two colors; this constraint is useful for constraining colors to be shades of one another.

B.2 Block Statics Model

In our statics model, blocks are assembled into structures via *contacts*: wherever two blocks touch, some internal force distribution arises over the resulting rectangular contact region. We represent this distribution with forces at each of the contact region's four vertices:

- f_n: a compressive force normal to the contact region.
- f_{t1} and f_{t2}: two friction forces tangent to the contact region.

f_n is bounded to be non-negative, and f_{t1} and f_{t2} are bounded to be within $|s \cdot f_n|$, where s is the coefficient of static friction of the contact. We use $s = 0.5$ for all results

presented in Chapter 4 unless noted otherwise. This statics model is essentially the same as that used by prior work on stable procedural buildings [116]. It is important to note that because friction force directions are treated as free variables, this model is not strictly physically accurate; correct handling of frictional contacts in a statics context is still a challenging problem [113].

Given this statics model, the process for generating a stable structure is straightforward. The user first writes a program that generates a random block structure, e.g. by iteratively stacking and perturbing random blocks. Our system then computes the net force $\bar{f}$ and net torque $\bar{\tau}$ on the center of mass of each block i and combines these 'residuals' into the following factor:

$$\sum_i \texttt{softeq}(||\bar{f}_i||, 0, \sigma_f) + \texttt{softeq}(||\bar{\tau}_i||, 0, \sigma_\tau)$$

The bandwidths σ_f and σ_τ must be set in an appropriately scale-invariant fashion, so that large structures are not penalized more than small ones. One option is to define them as percentages of the average external (i.e. due to gravity) force and torque acting on the structure. We find that 1% tolerance keeps an HMC sampler sufficiently close to the static equilibrium manifold while still allowing for exploration.

Appendix C

SOSMC Proof of Correctness

We aim to show that the marginal distribution over models generated by SOSMC is the same as the marginal distribution over models generated by depth-first SMC.

We will use the fact that a random choice variable r in a trace $\mathbf{r}_n$ can be uniquely addressed by its position in the function call tree of the trace [119]. We call this address $\text{addr}(r)$.

Definition 1. Two variables r^1 and r^2 are equivalent ($r^1 \equiv r^2$) if their addresses and values are the same. That is, $\text{addr}(r^1) = \text{addr}(r^2)$ and $r^1 = r^2$.

Definition 2. Two traces $\mathbf{r}_n^1$ and $\mathbf{r}_n^2$ are equivalent ($\mathbf{r}_n^1 \equiv \mathbf{r}_n^2$) if they contain equivalent variables. That is,

- $\forall r_{n,i}^1$, $\exists r_{n,j}^2$ such that $r_{n,i}^1 \equiv r_{n,j}^2$.
- $\forall r_{n,j}^2$, $\exists r_{n,i}^1$ such that $r_{n,j}^2 \equiv r_{n,i}^1$.

In particular, we assume that equivalent traces are considered equivalent by the scoring function $s(\cdot)$—that is, the score assigned to a trace does not depend upon the order in which it was generated.

Assumption 1. If $\mathbf{r}_n^1 \equiv \mathbf{r}_n^2$, then $s(\mathbf{r}_n^1) = s(\mathbf{r}_n^2)$. *

Together, these definitions allow us to group traces into equivalence classes $\mathbf{X}_n$, where all $\mathbf{x}_n = \{\mathbf{r}_n, \mathbf{o}_n\} \in \mathbf{X}_n$ generate the same partial model. Formally, our goal is to show that $P_N^{\pi_D}(\mathbf{X}_N) = P_N^{\pi_S}(\mathbf{X}_N)$. We start with defining the unnormalized density of an equivalence class by marginalizing out all the orderings that generate it. Let $\hat{\mathbf{r}}_n$ be any trace from equivalence class $\mathbf{X}_n$. Then:

$$
\begin{aligned}
F_n(\mathbf{X}_n) &= \sum_{\mathbf{x}_n \in \mathbf{X}_n} F_n(\mathbf{x}_n) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \cdot p_n(\mathbf{x}_n) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\mathbf{x}_m \setminus \mathbf{x}_{m-1}|} p(x_{m,i} | x_{m,1:(i-1)}, \mathbf{x}_{m-1}) \\
&= \sum_{\mathbf{x}_n \in \mathbf{X}_n} s(\mathbf{r}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\mathbf{r}_m \setminus \mathbf{r}_{m-1}|} p(r_{i,m} | \mathrm{par}(r_{i,m})) \\
&\qquad\qquad \prod_{m=1}^{n} \prod_{j=1}^{|\mathbf{o}_m \setminus \mathbf{o}_{m-1}|} \pi(o_{j,m} | \mathbf{x}_m) \\
&= s(\hat{\mathbf{r}}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\hat{\mathbf{r}}_m \setminus \hat{\mathbf{r}}_{m-1}|} p(\hat{r}_{i,m} | \mathrm{par}(\hat{r}_{i,m})) \\
&\qquad \left(\sum_{\mathbf{x}_n \in \mathbf{X}_n} \prod_{m=1}^{n} \prod_{j=1}^{|\mathbf{o}_m \setminus \mathbf{o}_{m-1}|} \pi(o_{j,m} | \mathbf{x}_m) \right) \\
&= s(\hat{\mathbf{r}}_n) \prod_{m=1}^{n} \prod_{i=1}^{|\hat{\mathbf{r}}_m \setminus \hat{\mathbf{r}}_{m-1}|} p(\hat{r}_{i,m} | \mathrm{par}(\hat{r}_{i,m})
\end{aligned}
$$

We can move the $s(\mathbf{r}_n) \cdots$ terms outside the summation because all $\mathbf{r}_n$ are equivalent (Definition 2, Assumption 1) and because the remaining terms—the ordering probabilities—form a discrete probability distribution whose elements sum to one.

By Equation 5.1, it remains to show that $Z_N^{\pi_D} = Z_N^{\pi_S}$. By the definition of partition

*Efficient, incrementalized implementations that use intermediate results of $s(\mathbf{r}_{n-1})$ to compute $s(\mathbf{r}_n)$ must guarantee this property.

function,

$$Z_N^\pi = \int_{\mathcal{X}^\pi} F_N(\mathbf{x})\mathrm{d}\mathbf{x} = \int_{\Omega^\pi} F_N(\mathbf{X})\mathrm{d}\mathbf{X}$$

where $\mathcal{X}^\pi$ is the set of all complete traces that can be generated under ordering policy π and Ω^π is the set of all equivalence classes (from here on, we omit the N subscript for brevity). Thus it suffices to show that $\Omega^{\pi_S} = \Omega^{\pi_D}$.

Lemma 1. $\Omega^{\pi_D} = \Omega^{\pi_S}$, *which means*

1. $\forall \mathbf{x}^D \in \mathcal{X}^{\pi_D}$, $\exists \mathbf{x}^S \in \mathcal{X}^{\pi_S}$ *such that* $\mathbf{r}^D \equiv \mathbf{r}^S$.
2. $\forall \mathbf{x}^S \in \mathcal{X}^{\pi_S}$, $\exists \mathbf{x}^D \in \mathcal{X}^{\pi_D}$ *such that* $\mathbf{r}^S \equiv \mathbf{r}^D$.

Proof.

1. $\forall \mathbf{x}^D \in \mathcal{X}^{\pi_D}$, $\mathbf{x}^D \in \mathcal{X}^{\pi_S}$, since the fixed ordering generated by π_D can be generated by π_S with nonzero probability.
2. $\forall \mathbf{x}^S \in \mathcal{X}^{\pi_S}$, create an empty trace $\mathbf{r}^D$ and walk the function call tree of $\mathbf{r}^S$ in depth-first order. When encountering a variable r with location in the call tree given by addr(r), insert that variable into $\mathbf{r}^D$. This process results in a valid trace in $\mathcal{X}^{\pi_D}$ which is equivalent to $\mathbf{r}^S$. □

We have proven that $P_N^{\pi_D}(\mathbf{X}_N) = P_N^{\pi_S}(\mathbf{X}_N)$ for programs that always terminate after at most N steps. Procedural models that explicitly limit recursion depth or that stop when geometric features become too small fit this description. Without such checks, procedural models only *almost* always terminate after a finite number of steps, i.e. termination probability approaches one as the number of steps approaches infinity. The same analysis should hold in this case as well, as probabilistic programs that terminate with probability one have well-defined marginal distributions over execution traces [29]. The proof would require a limit argument on N for approximating finite programs.

Bibliography

[1] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov Chain Monte Carlo Methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3), 2010.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[3] Fan Bao, Dong-Ming Yan, Niloy J. Mitra, and Peter Wonka. Generating and Exploring Good Building Layouts. In *SIGGRAPH 2013*.

[4] Bedřich Beneš, Ondvrej Šava, Radomir Měch, and Gavin Miller. Guided Procedural Modeling. In *Eurographics 2011*.

[5] R. Bergman. *Wood Handbook – Wood as an Engineering Material*. Forest Products Laboratory, 2010.

[6] R. Bisiani. Beam Search. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*. 1987.

[7] Craig Boutilier, Nir Friedman, Moises Goldszmidt, and Daphne Koller. Context-specific Independence in Bayesian Networks. In *UAI 1996*.

[8] Marcus A. Brubaker, Mathieu Salzmann, and Raquel Urtasun. A Family of MCMC Methods on Implicitly Defined Manifolds. In *AISTATS 2012*.

[9] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. In

ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, 2007.

[10] Swarat Chaudhuri and Armando Solar-Lezama. Smooth Interpretation. In *PLDI 2010*.

[11] Yan Chen, Joshua Dunfield, and Umut A. Acar. Type-Directed Automatic Incrementalization. In *PLDI 2012*.

[12] Stephen Chenney and D. A. Forsyth. Sampling Plausible Solutions to Multibody Constraint Problems. In *SIGGRAPH 2000*.

[13] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. Programmatic and Direct Manipulation, Together at Last. In *PLDI 2016*.

[14] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann. *Automatic Differentiation: From Simulation to Optimization*. Computer and Information Science. Springer, 2001.

[15] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 1989.

[16] Minh Dang, Stefan Lienhard, Duygu Ceylan, Boris Neubert, Peter Wonka, and Mark Pauly. Interactive Design of Probability Density Functions for Shape Grammars.

[17] Ana Lucia de Moura and Roberto Ierusalimschy. Revisiting Coroutines. Technical report, 2004.

[18] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: A Multi-stage Language for High-performance Computing. In *PLDI 2013*.

[19] Zachary DeVito, Michael Mara, Michael Zollhoefer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niener. Opt: A Domain Specific Language for Non-linear Least Squares Optimization in Graphics and Imaging. *CoRR*, arXiv:1604.06525, 2016.

[20] R. Douc and O. Cappe. Comparison of Resampling Schemes for Particle Filtering. In *ISPA 2005*.

[21] Arnaud Doucet, Nando De Freitas, and Neil Gordon, editors. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[22] Steven P. Dow, Alana Glassco, Jonathan Kass, Melissa Schwarz, Daniel L. Schwartz, and Scott R. Klemmer. Parallel Prototyping Leads to Better Design Results, More Divergence, and Increased Self-efficacy. *ACM Trans. Comput.-Hum. Interact.*, 17(4), 2010.

[23] Simon Duane, A.D. Kennedy, Brian J. Pendleton, and Duncan Roweth. Hybrid Monte Carlo. *Physics Letters B*, 195(2):216 – 222, 1987.

[24] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[25] S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. *CoRR*, arXiv:1603.08575, 2016.

[26] Shaohua Fan. *Sequential Monte Carlo Methods for Physically Based Rendering*. PhD thesis, 2006.

[27] Matthew Fisher, Daniel Ritchie, Manolis Savva, Thomas Funkhouser, and Pat Hanrahan. Example-based Synthesis of 3D Object Arrangements. In *SIGGRAPH Asia 2012*.

[28] Walter R. Gilks and Carlo Berzuini. Following a moving target—Monte Carlo inference for dynamic Bayesian models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 63(1), 2001.

[29] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI 2008*.

[30] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2015-12-23.

[31] N.J. Gordon, D.J. Salmond, and A.F.M. Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *Radar and Signal Processing, IEE Proceedings F*, 140(2), 1993.

[32] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical report, 1974.

[33] Shixiang Gu, Zoubin Ghahramani, and Richard E. Turner. Neural Adaptive Sequential Monte Carlo. In *NIPS 2015*.

[34] Brian Guenter. Efficient Symbolic Differentiation for Graphics Applications. In *SIGGRAPH 2007*.

[35] Robert H. Halstead, Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 1985.

[36] Perttu Hämäläinen, Sebastian Eriksson, Esa Tanskanen, Ville Kyrki, and Jaakko Lehtinen. Online Motion Synthesis Using Sequential Monte Carlo. In *SIGGRAPH 2014*.

[37] J. M. Hammersley and K. W. Morton. Poor Man's Monte Carlo. *Journal of the Royal Statistical Society. Series B (Methodological)*, 16(1), 1954.

[38] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *ISCA 1993*.

[39] Shawn Hershey, Jeffrey Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theophane Weber, and Benjamin Vigoda. Accelerating Inference: towards a full Language, Compiler and Hardware stack. *CoRR*, arXiv:1212.2991, 2012.

[40] Matthew D. Hoffman and Andrew Gelman. The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 2014.

[41] Qiang Huang, Jizhe Zhang, Arman Sabbaghi, and Tirthankar Dasgupta. Optimal Offline Compensation of Shape Shrinkage for 3D Printing Processes. *IIE Transactions on Quality and Reliability*.

[42] Irvin Hwang, Andreas Stuhlmüller, and Noah D. Goodman. Inducing Probabilistic Programs by Bayesian Program Merging. *CoRR*, arXiv:1110.5667, 2011.

[43] Interactive Data Visualization Inc. SpeedTree. Retrieved 2015-08-07 from `http://speedtree.com`, 2015.

[44] Singular Inversions. FaceGen. Retrieved 2015-07-27 from `http://facegen.com`, 2015.

[45] K. Norman J. Manning, R. Ranganath and D. Blei. Black Box Variational Inference. In *AISTATS 2014*.

[46] Arjun Jain, Thorsten Thormählen, Tobias Ritschel, and Hans-Peter Seidel. Material Memex: Automatic Material Suggestions for 3D Objects. In *SIGGRAPH Asia 2012*.

[47] Arjun Jain, Thorsten Thormählen, Tobias Ritschel, and Hans-Peter Seidel. Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination. *Comp. Graph. Forum*, 31(2pt3), 2012.

[48] Wenzel Jakob and Steve Marschner. Manifold exploration: a Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport. In *SIGGRAPH 2012*.

[49] D. Jansson and S. Smith. Design fixation. *Design studies*, 12(1), 1991.

[50] Evangelos Kalogerakis, Siddhartha Chaudhuri, Daphne Koller, and Vladlen Koltun. A Probabilistic Model for Component-based Shape Synthesis. In *SIGGRAPH 2012*.

[51] Robert E. Kass, Bradley P. Carlin, Andrew Gelman, and Radford M. Neal. Markov Chain Monte Carlo in Practice: A Roundtable Discussion. *The American Statistician*, 52(2), 1998.

[52] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR 2015*.

[53] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *ICLR 2014*.

[54] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[55] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *NIPS 2012*.

[56] Chinmay Kulkarni, Steven P Dow, and Scott R Klemmer. Early and repeated exposure to examples improves creative work. In *Design Thinking Research*. Springer, 2014.

[57] B. Leimkuhler and S. Reich. *Simulating Hamiltonian Dynamics*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2004.

[58] Roger P. Levy, Florencia Reali, and Thomas L. Griffiths. Modeling the effects of memory on human online sentence processing with particle filters. In *NIPS 2009*.

[59] J. P. Lewis. Generalized Stochastic Subdivision. *ACM Trans. Graph.*, 6(3), 1987.

[60] Ruy Ley-Wild, Matthew Fluet, and Umut A. Acar. Compiling Self-adjusting Programs with Continuations. In *ICFP 2008*.

[61] Lei Li, Yi Wu, and Stuart J. Russell. SWIFT: Compiled Inference for Probabilistic Programs. Technical report, EECS Department, University of California, Berkeley, 2015.

[62] Sharon Lin, Daniel Ritchie, Matthew Fisher, and Pat Hanrahan. Probabilistic Color-by-Numbers: Suggesting Pattern Colorizations Using Factor Graphs. In *SIGGRAPH 2013*.

[63] Fredrik Lindsten, Michael I. Jordan, and Thomas B. Schön. Particle Gibbs with Ancestor Sampling. *J. Mach. Learn. Res.*, 15(1), 2014.

[64] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, 2002.

[65] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, arXiv:1404.0099, 2014.

[66] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design Galleries: A General Approach to Setting Parameters for Computer Graphics and Animation. In *SIGGRAPH 1997*.

[67] A. Martinovic and L. Van Gool. Bayesian Grammar Learning for Inverse Procedural Modeling. In *CVPR 2013*.

[68] Andrew Mccallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *NIPS 2009*.

[69] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala, and Vladlen Koltun. Interactive Furniture Layout Using Interior Design Guidelines. In *SIGGRAPH 2011*.

[70] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. Equation of State Calculations by Fast Computing Machines. *The Journal of Computational Physics*, 21, June 1953.

[71] Brian Milch, Bhaskara Marthi, Stuart J. Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG: Probabilistic Models with Unknown Objects. In *IJCAI 2005*.

[72] Andriy Mnih and Karol Gregor. Neural Variational Inference and Learning in Belief Networks. In *ICML 2014*.

[73] Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. Recurrent Models of Visual Attention. In *NIPS 2014*.

[74] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural Modeling of Buildings. In *SIGGRAPH 2006*.

[75] Radomír Měch and Przemyslaw Prusinkiewicz. Visual Models of Plants Interacting with Their Environment. In *SIGGRAPH 1996*.

[76] Radford M. Neal. MCMC Using Hamiltonian Dynamics. *Handbook of Markov Chain Monte Carlo*, 2010.

[77] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. Color Compatibility From Large Datasets. *ACM Transactions on Graphics*, 2011.

[78] Li-Chen Ou and M Ronnier Luo. A colour harmony model for two-colour combinations. *Color Research & Application*, 2006.

[79] Brooks Paige and Frank Wood. A Compilation Target for Probabilistic Programming Languages. In *ICML 2014*.

[80] Vincent Pegoraro, Ingo Wald, and Steven G. Parker. Sequential Monte Carlo Adaptation in Low-Anisotropy Participating Media. *Computer Graphics Forum*, 27(4), 2008.

[81] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics, 2009.

[82] Martyn Plummer. JAGS: Just another Gibbs sampler. `http://mcmc-jags.sourceforge.net/`. Accessed: 2016-04-18.

[83] Romain Prévost, Emily Whiting, Sylvain Lefebvre, and Olga Sorkine-Hornung. Make It Stand: Balancing Shapes for 3D Fabrication. In *SIGGRAPH 2013*.

[84] P. Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag New York, Inc., 1990.

[85] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomír Měch. L-Systems: From The Theory To Visual Models Of Plants. In *CSIRO Symposium on Computational Challenges in Life Sciences*, 1996.

[86] Przemyslaw Prusinkiewicz, Mark James, and Radomír Měch. Synthetic Topiary. In *SIGGRAPH 1994*.

[87] G. Ramalingam and Thomas Reps. A Categorized Bibliography on Incremental Computation. In *POPL 1993*.

[88] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *ICML 2014*.

[89] Daniel Ritchie. Quicksand: A Lightweight Embedding of Probabilistic Programming for Procedural Modeling and Design. In *The 3rd NIPS Workshop on Probabilistic Programming*, 2014.

[90] Daniel Ritchie, Sharon Lin, Noah D. Goodman, and Pat Hanrahan. Generating Design Suggestions under Tight Constraints with Gradient-based Probabilistic Programming. In *Eurographics 2015*.

[91] Daniel Ritchie, Ben Mildenhall, Noah D. Goodman, and Pat Hanrahan. Controlling Procedural Modeling Programs with Stochastically-Ordered Sequential Monte Carlo. In *SIGGRAPH 2015*.

[92] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *AISTATS 2016*.

[93] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah D. Goodman. Neurally-Guided Procedural Models: Learning to Guide Procedural Models with Deep Neural Networks. *CoRR*, arXiv:1603.06143, 2016.

[94] G. O. Roberts, A. Gelman, and W. R. Gilks. Weak convergence and optimal scaling of random walk Metropolis algorithms. *The Annals of Applied Probability*, 7(1), 1997.

[95] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global Value Numbers and Redundant Computations. In *POPL 1988*.

[96] Marshall N. Rosenbluth and Arianna W. Rosenbluth. Monte Carlo Calculation of the Average Extension of Molecular Chains. *The Journal of Chemical Physics*, 23(2), 1955.

[97] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1. chapter Learning Internal Representations by Error Propagation. MIT Press, 1986.

[98] Michael Schwartz and Peter Wonka. Procedural Design of Exterior Lighting for Buildings with Complex Constraints. *ACM Transactions on Graphics*, 2014.

[99] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Sander Dieleman Marc Lanctot, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529, 2016.

[100] A. F. M. Smith and A. E. Gelfand. Bayesian Statistics without Tears: A Sampling-Resampling Perspective. *The American Statistician*, 46(2), 1992.

[101] Jeffrey Smith, Jessica Hodgins, Irving Oppenheim, and Andrew Witkin. Creating Models of Truss Structures with Optimization. In *SIGGRAPH 2002*.

[102] Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Champaign, IL, USA, 1980.

[103] David J Spiegelhalter, Andrew Thomas, Nicky Best, Wally Gilks, and D Lunn. BUGS: Bayesian inference using Gibbs sampling. `http://www.mrc-bsu.cam.ac.uk/bugs`. Accessed: 2016-04-18.

[104] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.5.0*, 2014.

[105] K.O. Stanley and J. Lehman. *Why Greatness Cannot Be Planned: The Myth of the Objective*. Springer International Publishing, 2015.

[106] O. Stava, S. Pirk, J. Kratt, B. Chen, R. Měch, O. Deussen, and B. Benes. Inverse Procedural Modelling of Trees. *Computer Graphics Forum*, 33(6), 2014.

[107] Jacob Steinhardt and Percy Liang. Filtering with Abstract Particles. In *ICML 2014*.

[108] Leland Stewart and Perry McCarty, Jr. Use of Bayesian belief networks to fuse continuous and discrete information for target recognition, tracking, and situation assessment. *SPIE*, 1992.

[109] Andreas Stuhlmüller, Robert X.D. Hawkins, N. Siddharth, and Noah D. Goodman. Coarse-to-Fine Sequential Monte Carlo for Probabilistic Programs. *CoRR*, arXiv:1509.02962, 2015.

[110] Nervous System. Kinematics Collection. Retrieved 2015-07-27 from `http://n-e-r-v-o-u-s.com/shop/line.php?code=15`, 2015.

[111] Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. Learning Design Patterns with Bayesian Grammar Induction. In *UIST 2012*.

[112] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis Procedural Modeling. *ACM Trans. Graph.*, 30(2), 2011.

[113] Nobuyuki Umetani, Takeo Igarashi, and Niloy J. Mitra. Guided Exploration of Physically Valid Shapes for Furniture Design. In *SIGGRAPH 2012*.

[114] Carlos A. Vanegas, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Paul Waddell. Inverse Design of Urban Procedural Models. In *SIGGRAPH Asia 2012*.

[115] Ondřej Šťava, Bedřich Beneš, Matthew Brisbin, and Jaroslav Křivánek. Interactive Terrain Modeling Using Hydraulic Erosion. In *SCA 2008*.

[116] Emily Whiting, John Ochsendorf, and Frédo Durand. Procedural Modeling of Structurally-Sound Masonry Buildings. In *SIGGRAPH Asia 2009*.

[117] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[118] David Wingate, Noah D. Goodman, Andreas Stuhlmüller, and Jeffrey M. Siskind. Nonstandard Interpretations of Probabilistic Programs for Efficient Inference. In *NIPS 2011*.

[119] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS 2011*.

[120] David Wingate and Theophane Weber. Automated Variational Inference in Probabilistic Programming. In *NIPS 2012 Workshop on Probabilistic Programming*.

[121] F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *AISTATS 2014*.

[122] F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. *CoRR*, arXiv:1507.00996, 2015.

[123] Michael Wörister, Harald Steinlechner, Stefan Maierhofer, and Robert F. Tobler. Lazy Incremental Computation for Efficient Scene Graph Rendering. In *HPG 2013*.

[124] Kai Xu, Hao Zhang, Daniel Cohen-Or, and Baoquan Chen. Fit and Diverse: Set Evolution for Inspiring 3D Shape Galleries. In *SIGGRAPH 2012*.

[125] Lingfeng Yang. *From Execution Traces to Specialized Inference*. PhD thesis, Stanford, CA, USA, 2015.

[126] Lingfeng Yang, Pat Hanrahan, and Noah D. Goodman. Generating Efficient MCMC Kernels from Probabilistic Programs. In *AISTATS 2014*.

[127] Yong-Liang Yang, Yi-Jun Yang, Helmut Pottmann, and Niloy J. Mitra. Shape Space Exploration of Constrained Meshes. In *SIGGRAPH Asia 2011*.

[128] Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. Synthesis of Tiled Patterns Using Factor Graphs. *ACM Trans. Graph.*, 32(1), 2013.

[129] Yi-Ting Yeh, Lingfeng Yang, Matthew Watson, Noah D. Goodman, and Pat Hanrahan. Synthesizing Open Worlds with Constraints Using Locally Annealed Reversible Jump MCMC. In *SIGGRAPH 2012*.

[130] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan, and Stanley J. Osher. Make It Home: Automatic Optimization of Furniture Arrangement. In *SIGGRAPH 2011*.

[131] Lap-Fai Yu, Sai-Kit Yeung, Demetri Terzopoulos, and Tony F. Chan. DressUp!: Outfit Synthesis Through Automatic Optimization. In *SIGGRAPH Asia 2012*.

[132] Mehmet Ersin Yumer, Paul Asente, Radomir Mech, and Levent Burak Kara. Procedural Modeling Using Autoencoder Networks. In *UIST 2015*.

[133] Lifeng Zhu, Weiwei Xu, John Snyder, Yang Liu, Guoping Wang, and Baining Guo. Motion-guided Mechanical Toy Modeling. In *SIGGRAPH Asia 2012*.

[134] Matthew Zucker, Nathan Ratliff, Anca Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher Dellin, J. Andrew (Drew) Bagnell, and Siddhartha Srinivasa. CHOMP: Covariant Hamiltonian Optimization for Motion Planning. *International Journal of Robotics Research*, May 2013.