Differentiable Visual Computing

by

Tzu-Mao Li

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

© Massachusetts Institute of Technology 2019. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 10, 2019

Certified by .....
Frédo Durand
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Differentiable Visual Computing

by

Tzu-Mao Li

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2019, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Derivatives of computer graphics, image processing, and deep learning algorithms have tremendous use in guiding parameter space searches, or solving inverse problems. As the algorithms become more sophisticated, we no longer only need to differentiate simple mathematical functions, but have to deal with general programs which encode complex transformations of data. This dissertation introduces three tools, for addressing the challenges that arise when obtaining and applying the derivatives for complex graphics algorithms.

Traditionally, practitioners have been constrained to composing programs with a limited set of coarse-grained operators, or hand-deriving derivatives. We extend the image processing language Halide with reverse-mode automatic differentiation, and the ability to automatically optimize the gradient computations. This enables automatic generation of the gradients of arbitrary Halide programs, at high performance, with little programmer effort. We demonstrate several applications, including how our system enables quality improvements of even traditional, feed-forward image processing algorithms, blurring the distinction between classical and deep learning methods.

In 3D rendering, the gradient is required with respect to variables such as camera parameters, light sources, geometry, and appearance. However, computing the gradient is challenging because the rendering integral includes visibility terms that are not differentiable. We introduce, to our knowledge, the first general-purpose differentiable ray tracer that solves the full rendering equation, while correctly taking the geometric discontinuities into account. We show prototype applications in inverse rendering and the generation of adversarial examples for neural networks.

Finally, we demonstrate that the derivatives of light path throughput, especially the second-order ones, can also be useful for guiding sampling in forward rendering. Simulating light transport in the presence of multi-bounce glossy effects and motion in 3D rendering is challenging due to the high-dimensional integrand and narrow high-contribution areas. We extend the Metropolis Light Transport algorithm by adapting to the local shape of the integrand, thereby increasing sampling efficiency. In particular, the Hessian is able to capture the strong anisotropy of the integrand. We use ideas from Hamiltonian Monte Carlo and simulate physics in Taylor expansion to draw samples from high-contribution region.

Thesis Supervisor: Frédo Durand
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to acknowledge by briefly reflecting how I ended up writing this dissertation.

I was fascinated by computer science since I knew the existence of computers. I love the idea of automating tedious computation and the intellectual challenges involved in the process of automation. Naturally, I enrolled in a computer science program during my undergraduate education. At there I was introduced to the enchanting world of computer graphics, where people are able to generate beautiful images with equations and code, instead of pen and paper. During my undergraduate and master's studies at National Taiwan University, I worked with Yung-Yu Chuang, the professor who brought me into computer graphics research. I started to read academic papers, and was mesmerized by people who contribute their own ideas to improve image generation. In the end I was also able to contribute my own little idea, by publishing a paper related to denoising Monte Carlo rendering images during my Master's study.

I hoped to do more, and decided to pursue a Ph.D. next. Among all the academic literature I studied, a few names caught my eyes. My Ph.D. advisor Frédo Durand is one of them. His work on frequency analysis for light transport simulation provides a rigorous and insightful theoretical foundation for sampling and reconstruction in light transport. I am also amazed by his versatility in research. At the point this dissertation was written, he has already worked on physically-based rendering, non-photorealistic rendering, computational photography, computer vision, geometric and material modeling, human-computer interaction, medical imaging, and programming systems. After I first met him, I also found out that he has a great sense of humor and is in general a very likable and good-tempered person.

I joined the MIT graphics group as a rendering person. Frédo and I decided that research on Metropolis light transport [216] aligns our interests the most. Metropolis light transport is a classical rendering algorithm that was recently revitalized thanks to Wenzel Jakob's reimplementation of this notoriously difficult-to-implement algorithm. We found the classical literature of Langevin Monte Carlo [181] and Hamiltonian Monte Carlo [48], and thought that the derivatives information these algorithms use can also help Metropolis light transport. I also learned about the field of automatic differentiation, which later became the most essential tool of this thesis. Frédo, Luke Anderson, and Shuang Zhao had a lot of discussions with me on this topic, which helped to shape my thoughts. Frédo also brought Ravi Ramamoorthi, Jaakko Lehtinen, and Wenzel Jakob into this project. Ravi helped the most on this particular project. When we were collaborating, every week he would monitor my progress, and try to understand the current obstacles and provide advice. While the discussions with Frédo and others are usually higher-level, Ravi tried to understand every single detail of what I

do. Explaining my thought process to them clarified my thinking. Ravi and Frédo also have strong influences on my paper writing style. Frédo taught me to focus on the high-level pictures and Ravi taught me to explain everything in a clear manner.

Like many other computer science research projects, this project ended up to be a huge undertaking of software engineering. Our algorithm requires the Hessian and gradient of the light transport contribution. Implementing this efficiently requires complex metaprogramming and is very difficult with existing tools. This also motivates my later work on extending the Halide programming language [171] for generating gradients. Nevertheless, I was still able to implement the first differentiable bidirectional path tracer that is able to generate gradient and Hessian of path contribution. We published a paper in 2015.

The results from our first project on improving Metropolis light transport were quite encouraging. This motivated us to look deeper into this subfield. Our derivative-based algorithm helped local exploration, but the bigger issue of these Markov chain Monte Carlo methods lies in the global exploration. As light transport contribution is inherently multimodal due to discontinuities, the Markov chains in Metropolis light transport algorithms usually have bad mixing. This typically manifests as blotchy artifacts on the images. We were hoping to have a better understanding of the global structure of light transport path space, in order to resolve the exploration issue. Unfortunately, this turns out to be more difficult than we imagined, due to the curse of dimensionality. As the dimensionality of the path space increases, the difficulty to capture the structure increases exponentially. We were able to get decent results by fitting Gaussians on low-dimensional cases (say, 4D), but I got stuck as soon as I proceeded to higher-dimensional space.[1]

I stuck on the global Metropolis light transport project for nearly two years. In the meantime, I also explored a few other directions. For example, I tried to generalize gradient domain rendering [127] to the wavelet domain. None of these attempts were very successful. As most researchers already knew, when working on a research project, it is very difficult to know whether the researcher is missing something, or the project simply will go nowhere in the first place. Still, I gained a lot of useful knowledge in these years. During this period, I helped Luke Anderson on his programming system for rendering, Aether [7]. The system stores the Monte Carlo sampling process symbolically, and automatically produces the probability density function of this process. Like my first Metropolis rendering project, this process also heavily involves metaprogramming and huge engineering efforts. This increased my interests in systems research – many computer graphics researches are so engineering heavy,

[1] Recently, Reibold et al. [175] published a similar idea. A key feature that makes their idea works, in my opinion, is that they focus on fitting block-tridiagonal covariance matrices for their Gaussian mixtures, instead of the full covariance as we tried. This makes their problem significantly more tractable.

that we need better tools to help researchers and engineers for fast prototyping. I also did an internship at Weta digital to work with the Manuka rendering team, including Marc Droske, Jirí Vorba, Jorge Schwarzhaupt, Luca Fascione. I also met Lingqi Yan, who was also an intern there working on appearance modeling. We often chatted about research and video games together. The internship at Weta taught me a lot about production rendering, visual effects practices, and the beauty of New Zealand.

After I stagnated on the projects for a while, to avoid sunk cost fallacy, Frédo and I decided to temporarily move on. Inspired by the recent success of deep learning, and my frustration on the lack of tools for general and efficient automatic differentiation, we chose to work on automatic differentiating Halide code. We picked Halide because it was developed by our group, so we are sufficiently familiar with it. Halide also strikes the balance between having a more general computation model than most deep learning frameworks, and focused enough for us to optimize the gradient code generation. We contacted Jonathan Ragan-Kelley and Andrew Adams, the parents of Halide, who also had the idea of adding automatic differentiation to Halide for a long time. We also brought in my labmate Michaël Gharbi, who is one of the most knowledgable and likable people in the world, to work on this. I learned a lot about deep learning and data-driven computing from Michaël and a lot about parallel programming systems from Jonathan and Andrew. It was super fun working with these people. I was also happy that my knowledge of automatic differentiation became useful in fields outside of rendering.

At the summer of 2017, I did an internship at Nvidia research at Seattle. Since we published the 2015 paper on improving Metropolis light transport using derivative information, we were always thinking about using it also for inverse rendering. Previous work focused on either simplified model [137] or volumetric scattering [62], and we think there are interesting cases on inverse surface light transport where derivatives can be useful. My collaborator and friend Jaakko saw my internship as an opportunity for pushing me to work in this direction. He also pointed out that the main technical challenge in the surface light transport case would be the non-differentiability. Under the help of Jaakko, Marco Salvi, and Aaron Lefohn, we experimented several different approximation algorithms there. They work well in many cases, but there were always some edge cases that would break the algorithms. The time at Nvidia was a fresh break from graduate school. I met Lingqi again there, and Christoph Peters, another intern who has an unusual passion for the theory of moment reconstruction problems, and always have only apples for his lunch. I also met two other interns Qi Sun, Tri Nguyen and maintained friendships with them since then.

After I wrapped up my internship at Nvidia and returned to MIT, I and Jaakko talked with Frédo and Miika Aittala about the inverse rendering project. Frédo pointed out the relation

between the mesh discontinuities and silhouette rasterization [190]. After more discussions, we realized that this is highly related to Ravi's first-order analysis work back in 2007 [173]. I worked out the math and generalized Ravi's theory to arbitrary parameters, primary visibility, and global illumination. Based on my experience on the differentiable bidirectional path tracer from 2015, I was able to quickly come up with a prototype renderer and to write a paper about this.

This is how this dissertation was written.

Since I mostly focused on the research part of the story, many people were left out. I am grateful to the labmates in MIT graphics group. Lukas Murmann and Alexandre Kaspar were the students who joined the group at the same time as me. Naturally, we hung out a lot together (by my standard). YiChang Shih, Abe Davis, and Valentina Shin are the senior students who guided us when we were lost. Zoya Bylinski made sure we always have enough snacks and coffee in the office. I enjoyed all the trash-talks with Tiam Jaroensri. Prafull Sharma's jokes are sometimes funny, sometimes not too much. Camille Biscarrat and Caroline Chan brought energy and fresh air to our offices. I had a lot of nerdy programming languages chat with Yuanming Hu. It was also fun to chat about research with Gaurav Chaurasia, Aleksandar Zlateski, and David Levin. Luke Anderson proofread every single paper I have written, including this dissertation. Nathaniel Jones broadened my view on rendering's application in architectural visualization.

The reviewer #1 of the inverse rendering paper, who I suspect to be Ravi, provided an extremely detailed and helpful review. My thesis committee Justin Solomon and Wojciech Matusik also provided useful comments. Justin pointed out the relation between the Reynold transport theorem and our edge sampling method in the inverse rendering project. I should also thank NSF and Toyota Research Institute for the funding support, so I don't freeze to death in Boston.

Anton Kaplanyan invited me to intern at Facebook Reality Lab during 2018. I met Thomas Leimkühler, Steve Bako, Christoph Schied, and Michael Doggett there. FRL was a very competitive and vigorous environment. I loved the free food. They had smoked salmon for the breakfast! I worked with Dejan Azinović, Matthias, Nießner and Anton there on a material and lighting reconstruction project.

I met my girlfriend Ailin Deng at the end of 2017. She has since then become the oasis that shelters me when I am tired of programming and research. I thank I-Chao Shen and Sheng-Chieh Chin for being patient for listening to my rants and non-sensical research ideas. I am grateful that my parents remain supportive throughout my studies on computer science.

Contents

1 Introduction

Differential calculus seeks to characterize the local geometry of a function. By definition, the derivatives at a point of a function tell us what happens to the outputs if we slightly move the point. This property enables us to make smarter decisions, and makes derivatives a fundamental tool for various tasks including parameter tuning, solving inverse problems, and sampling.

As computer graphics and image processing algorithms become more sophisticated, computing derivatives for functions defined in these algorithms becomes more important. Derivatives are useful in both data-driven and non data-driven scenarios. For one thing, as the number of parameters of the algorithm increases, it becomes infeasible to manually adjust them to achieve the desired behavior. Data-driven approaches allow us to automatically tune the parameters of our model. For another thing, these sophisticated *forward* models can be used for solving *inverse* problems. For example, the computer graphics community has developed mature models of how photons interact with scenes and cameras, and it is desirable to incorporate this knowledge, instead of learning it from scratch using a data-driven approach. Finally, differentiable algorithms are *composable*, which means we can piece different differentiable algorithms together, and have an end-to-end differentiable system as a whole. This enables us to compose novel algorithms by adding other differentiable components to the pipeline, such as deep learning architectures. Figure 1-1 illustrates the use of derivatives.

While deep learning has popularized the use of gradient-based optimization over highly-parameterized functions, the current building blocks used in deep learning methods are very limited. A typical deep learning architecture is usually composed of convolution filters, linear combinations of elements ("fully connected" layer), subsampling by an integer factor ("pooling", usually by a factor of 2), and element-wise nonlinearities. Most visual computing algorithms are far more sophisticated than these. They often combine neighboring pixels using non-linear kernels (e.g. [209, 28]), downsample a signal prefiltered by some antialiasing filters (e.g. [150]), use heavy-tailed non-linear functions to model the reflectance of surfaces (e.g. [43]), or traverse trees for finding intersections between objects (e.g. [40]).

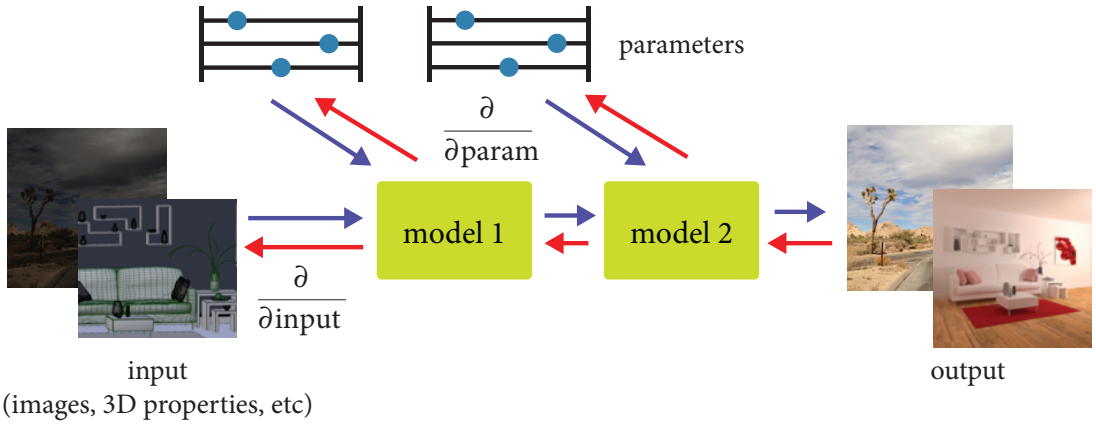


Figure 1-1: **Differentiable visual computing.** Derivatives enable us to make smart decisions in our models. The derivatives of a model's output can be taken with respect to the parameters or the inputs of the model. This makes it possible to find corresponding inputs for a given output, solving an inverse problem, or we can find model parameters that map inputs to outputs. This is useful in both data-driven and non-data-driven applications.

I argue that most numerical algorithms in computer graphics and image processing should be implemented in a differentiable manner. This is beneficial for both data-driven and non-data-driven applications. Comparing to deep learning approaches, this allows better control and interpretability by integrating the domain knowledge into the model. It makes debugging models a lot easier since we have a better idea of how data should interact with the model. It is often more efficient both in time and memory, and more accurate, since the model is more tailored to the applications.

Efficiently evaluating derivatives from algorithms that perform complex transformations on 3D data or 2D images presents challenges in both systems and algorithms. Firstly, existing deep learning frameworks (e.g. [1, 165]) only have limited expressiveness. While automatic differentiation methods (e.g. [71]) can generate derivatives from almost arbitrary algorithms, generating efficient derivative code while taking parallelism and locality into consideration is still difficult. Secondly, the algorithms can introduce discontinuities. For example, in 3D rendering, the visibility term is discontinuous, which prevents direct application of automatic differentiation. Finally, designing algorithms that efficiently utilize the obtained derivatives is also important.

The contributions of this dissertation are three novel tools for addressing the challenges and for investigating the use of derivatives in the context of visual computing.

Efficient Automatic Differentiation for Image Processing and Deep Learning In Chapter 4, we address the systems challenges for efficiently generating derivatives code from image processing algorithms. Existing tools for automatically generating derivatives have at least one of the following two issues:

- General automatic differentiation systems (e.g. [22, 69, 77, 86, 229]) are *inefficient* because they do not take parallelism and locality into consideration.

- Deep learning frameworks (e.g. [1, 165]) are *inflexible* because they are composed of coarse-grained, specialized operators, such as convolutions or element-wise operations. For many applications, it is often difficult to assemble these operators to build the desired algorithm. Even when done successfully, the resulting code is often both slow and memory-inefficient, saving and reloading entire arrays of intermediate results between each step, causing costly cache misses.

These limitations are one of the main obstacles preventing researchers and developers from inventing novel differentiable algorithms, since they are often required to manually derive and implement the derivatives in lower-level languages such as C++ or CUDA.

In this chapter, we focus on image processing and deep learning. We build on the image processing language Halide [171, 172], and extend it with the ability to generate gradient code. Halide provides a concise and natural language for expressing image processing algorithms, while allowing the separation between high-level algorithm and low-level scheduling for achieving high-performance across platforms. To generate efficient gradient code, we develop a compiler transformation for generating gradient code automatically from Halide algorithms. Keys to making the transformation work are a scatter-to-gather conversion algorithm which preserves parallelism, and a simple automatic scheduling algorithm which specializes in the patterns in gradient code and provides a GPU backend.

Using this new extension of Halide, we show first that we can concisely and efficiently implement existing custom deep learning operators, which previously required implementation in low-level CUDA. Our generated code is as fast or even faster than the corresponding high-performance hand-written code, with less than 1/10 of the lines of code. Secondly, we show that gradient-based parameter optimization is useful outside of traditional deep learning approaches. We significantly improve the accuracy of two traditional image processing algorithms by augmenting their parameters and automatically optimizing them. Thirdly, we show that the system is also useful for inverse problems. We implement a novel joint burst demosaicking and superresolution algorithm by building a forward image formation model. Finally, we demonstrate our extension's versatility by implementing two applications

outside of image processing – lens design optimization through a ray tracer and a classical fluid simulator in computer graphics [201].

Differentiable Monte Carlo Ray Tracing through Edge Sampling While automatic differentiation generates derivatives, it does not handle non-differentiability in individual code paths. In particular, for computer graphics, we are interested in the gradients of the 3D rendering operation with respect to variables such as camera parameters, light sources, scene geometry, and appearance. While the rendering integral is differentiable, the *integrand* is discontinuous due to visibility. Previous works on differentiable rendering (e.g. [137, 109]) focused on fast approximate solutions, and do not handle secondary effects such as shadows or global illumination.

In Chapter 5, we introduce a general-purpose differentiable ray tracer, which, to our knowledge, is the first comprehensive solution that is able to compute the gradients of the rendering integral with respect to scene parameters, while correctly taking geometric discontinuities into consideration. We observe that the discontinuities in the rendering integral become Dirac delta functions when taking the gradient. Therefore we develop a novel method for explicit sampling of the triangle edges that introduce the discontinuities. This requires new spatial acceleration techniques and importance sampling for efficiently selecting edges.

We integrate our differentiable ray tracer with the automatic differentiation library PyTorch [165], and demonstrate prototype applications for inverse rendering and finding adversarial examples for neural networks.

Hessian-Hamiltonian Monte Carlo Rendering Finally, we show that derivatives, especially the second-order ones, can also be used for accelerating forward rendering by guiding light path sampling. In Chapter 6, we present a Markov chain Monte Carlo rendering algorithm that automatically and explicitly adapts to the local shape of the integrand using the second-order Taylor expansion, thereby increasing sampling efficiency. In particular, the Hessian is able to capture the strong anisotropy caused by challenging effects such as multi-bounce glossy effects and motion.

Using derivatives in the context of sampling instead of optimization requires more care. The second-order Taylor expansion does not define a proper distribution, and therefore cannot be directly importance sampled. We use ideas from Hamiltonian Monte Carlo [48] that simulates Hamiltonian dynamics in a flipped version of the Taylor expansion where gravity pulls particles towards the high-contribution region. The quadratic landscape leads to a closed-form anisotropic Gaussian distribution, and results in a standard Metropolis-

Hastings algorithm [78].

Unlike previous works that derive the sampling procedures manually and only consider specific effects, our resulting algorithm is general thanks to automatic differentiation. In particular, our method is the first Markov chain Monte Carlo rendering algorithm that is able to resolve the anisotropy in the time dimension and render difficult moving caustics.

1.1 Background and Target Audience

Chapter 2 and Chapter 3 review the background of automatic differentiation, optimization, and sampling, and their relationship. These are not novel components of this dissertation, but they represent important components that are glossed over in the individual publications. Moreover, they connect the central themes of this dissertation: differentiating algorithms and making use of the resulting derivatives.

I imagine the majority of readers of this dissertation to be researchers in the fields of computer graphics, image processing, systems or machine learning, who are interested in using the individual tools and want to know the details better, or people who are building their own differentiable systems. For both groups of people, I hope the examples in this dissertation can improve your intuition on building differentiable systems in the future.

1.2 Publications

The content of this dissertation has appeared in the following publications:

- Chapter 4: Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018
- Chapter 5 Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018
- Chapter 6 Tzu-Mao Li, Jaakko Lehtinen, Ravi Ramamoorthi, Wenzel Jakob, and Frédo Durand. Anisotropic Gaussian mutations for Metropolis light transport through Hessian-Hamiltonian dynamics. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 34(6):209:1–209:13, 2015

The source code of the projects can be downloaded from the corresponding project sites:

- http://gradients.halide-lang.org/

- https://people.csail.mit.edu/tzumao/diffrt/
- https://people.csail.mit.edu/tzumao/h2mc/

For Chapter 4, I added a hindsight on differentiating scan operations (Chapter 4.3.2) and an example of fluid simulation (Chapter 4.4.4) since the publication.

For Chapter 5, I added more discussions about the pathelogical parallel edges condition where our method can produce incorrect result (Chapter 5.2). There is also some discussions regarding non-linear camera models (Chapter 5.2). I added some discussion related to Reynolds transport theorem and shape optimization (Chapter 5.2.4). I also revised the edge selection algorithm (Chapter 5.3.1), added some discussions about a GPU implementation (Chapter 5.4), and added discussions about differentiable geometry buffer rendering (Chapter 5.4.3).

For Chapter 6, I added a description of an improved large step mutation method (Chapter 6.4), and some discussions to recent works (Chapter 6.5.1).

2 | Automatic Differentiation

Evaluating derivatives for computer graphics and image processing algorithms is the key to this dissertation. We will use them to minimize cost functions, solve inverse problems, and guide sampling procedures. Intuitively speaking, the derivatives of a function characterize the local behavior at a given point, e.g. if I move the point to this direction, will the output values become larger or smaller? This allows us to find points that result in certain function values, such as maximizing a utility function, or minimizing the difference between the output and a target.

In this chapter, we review the methods for generating derivatives from numerical programs. The chapter serves as an introductory article to the theory and practice of automatic differentiation. The reader is encouraged to read Griewank and Walther's textbook [71] for a comprehensive treatment of the topic.

Given a computer program containing control flow, loops, and/or recursion, with some real number inputs and some real number outputs, our goal is to compute the derivatives between the outputs and the inputs. Sometimes there is only a scalar output but more than one input, in which case we are interested in the gradient vector. Sometimes there are multiple outputs as well, and we are interested in the Jacobian matrix. Sometimes we are interested in the higher-order derivatives such as the Hessian matrix.

While the title of this chapter is *automatic* differentiation, we will also talk about how to differentiate a program *manually*, which is less difficult than one might imagine. We show how to systematically write down the derivative code just by looking at a program, without lengthy and convoluted mathematical notation. While this is still more tedious and error-prone than an automatic compiler transformation (which is why we develop the tool in Chapter 4), it is a useful practice for understanding the structure of derivative code, and is even practical sometimes if it is difficult to parse and transform the code.

2.1 Finite Differences and Symbolic Derivatives

Before discussing automatic differentiation algorithms, it is useful to review other ways of generating derivatives, and compare them to automatic differentiation.

A common approximation for derivatives are finite differences, sometimes also called numerical derivatives. Given a function $f(x)$ and an input x, we approximate the derivative by perturbing x by a small amount h:

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x)}{h} \quad \text{or} \tag{2.1}$$

$$\frac{df(x)}{dx} \approx \frac{f(x+h) - f(x-h)}{2h}. \tag{2.2}$$

The problem with this approximation is two-fold. First, the optimal choice of the step size h in a computer system is problem dependent. If the step size is too small, the rounding error of the floating point representation becomes too large. On the other hand, if the step size is too large, the result becomes a poor approximation to the true derivative. Second, the method is inefficient for multivariate functions. For a function with 100 variables and a scalar output, computing the full gradient vector requires at least 101 evaluations of the original function.

Another alternative is to treat the content of the function f as a sequence of mathematical operations, and symbolically differentiate the function. Indeed, most of the rules for differentiation are mechanical, and we can apply the rules to generate $f'(x)$. However, in our case, $f(x)$ is usually an *algorithm*, and symbolic differentiation does not scale well with the number of symbols. Consider the following code:

```
function f(x):
    result = x
    for i = 1 to 8:
        result = exp(result)
    return result
```

Figure 2-1: A code example that iteratively computes a nested exponential for demonstrating the difference between symbolic differentiation and automatic differentiation.

Using the symbolic differentiation tools from mathematical software such as Mathematica [93] would result in the following expression:

$$\frac{df(x)}{dx} = e^{x+e^{e^{e^{e^{e^{e^{e^{e^x}}}}}}}} + e^{e^{e^{e^{e^{e^{e^{e^x}}}}}}} + e^{e^{e^{e^{e^{e^{e^x}}}}}} + e^{e^{e^{e^{e^{e^x}}}}} + e^{e^{e^{e^{e^x}}}} + e^{e^{e^{e^x}}} + e^{e^{e^x}} + e^{e^x}. \tag{2.3}$$

The size of derivative expression will become intractable when the size of the loop grows

much larger. Using forward-mode automatic differentiation, which will be introduced later, we can generate the following code for computing derivatives:

```
function d_f(x):
    result = x
    d_result = 1
    for i = 1 to 8:
        result = exp(result)
        d_result = d_result * result
    return d_result
```

The code above outputs the exact same values as the symbolic derivative (Equation 2.3), but is significantly more efficient (8 v.s. 37 exponentials). This is due to automatic differentiation's better use of the intermediate values and the careful factorization of common subexpressions.

2.2 Algorithms for Generating Derivatives

For a better understanding of automatic differentiation, before introducing the fully automatic solution, we will first discuss how to manually differentiate a code example. We start from programs with only function calls and elementary operations such as addition and multiplication. In particular, we do not allow recursive or circular function calls. Later in Chapter 2.3.1, we generalize the idea to handle control flow such as loops and branches, and handle recursion. Throughout the chapter, we assume all function calls are side-effect free. To the author's knowledge, there are no known automatic differentiation algorithms for transforming arbitrary functions with side effects.

The key to automatic differentiation is the chain rule. Consider the following code with input `x` and output `z`:

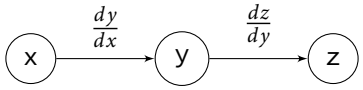
```
y = f(x)
z = g(y)
```

Assume we already know the derivative functions $\frac{df(x)}{dx}$ and $\frac{dg(y)}{dy}$, and we are interested in the derivative of the output `z` with respect to input `x`. We can compute the derivative by applying the chain rule:

```
dydx = dfdx(x)
dzdy = dgdy(y)
dzdx = dzdy * dydx
```

We can recursively apply the rule to generate derivative functions, until the function is an elementary function for which we know the analytical derivatives, such as addition, multiplication, `sin()`, or `exp()`.

A useful mental model for automatic differentiation is the *computational graph*. It can be used for representing dependencies between variables. The nodes of the graph are the variables and the edges are the derivatives between the adjacent vertices. In the case above the graph is linear:



Computing derivatives from a computational graph involves traversal of the graph, and gathering of different paths that connect inputs and outputs.

In practice, most functions are multivariate, and often times we want to have multiple derivatives such as for the gradient vector. In this case, different derivatives may have common paths in the computational graph that can be factored out, which can greatly impact efficiency. Consider the following code example and its computational graph:

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)
```

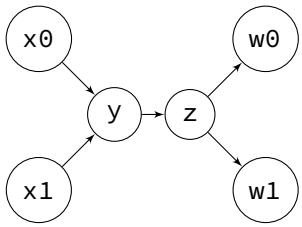
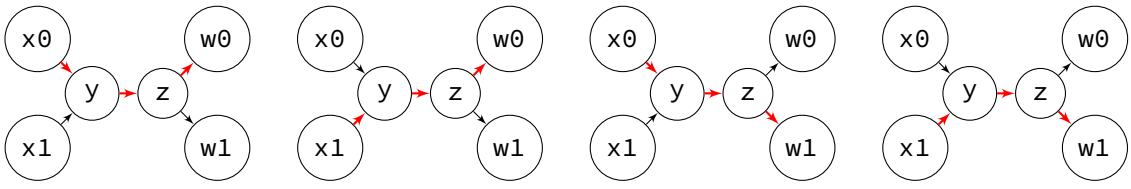


Figure 2-2: Code example and computational graph with two inputs `x0, x1` and two outputs `w0, w1`

There are four derivatives between the two outputs and two inputs. We can obtain them by traversing the four corresponding paths in the computational graph:



(a) $\frac{\partial w_0}{\partial x_0}$ (b) $\frac{\partial w_0}{\partial x_1}$ (c) $\frac{\partial w_1}{\partial x_0}$ (d) $\frac{\partial w_1}{\partial x_1}$

For example, in (a), the derivative of `w0` with respect to `x0` is the product of the three red edges:

$$\frac{\partial w_0}{\partial x_0} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x_0}, \tag{2.4}$$

and in (b), the derivative of `w0` with respect to `x1` is

$$\frac{\partial w_0}{\partial x_1} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x_1}. \tag{2.5}$$

We can observe that some of the derivatives share common subpaths in the computational graph. For example the two derivatives above $\frac{\partial w_0}{\partial x_0}$ and $\frac{\partial w_0}{\partial x_1}$ share the same subpath `y, z, w0`. We can therefore factor this subpath out and premultiply $\frac{\partial w_0}{\partial y} = \frac{\partial w_0}{\partial z}\frac{\partial z}{\partial y}$ for the two derivatives. In a larger computational graph, this factorization can have enormous impact on the performance of the derivative code, even affecting the time complexity in terms of the number of inputs or outputs.

Different automatic differentiation algorithms find common factors in the computational graph in different ways. In the most general case, finding a factorization that results in minimal operations is NP-hard [154]. Fortunately, in many common cases, such as factorization for the gradient vector, there are efficient solutions.[1]

If the input is a scalar variable, no matter how many variables there are in the output, *forward-mode* automatic differentiation generates derivative code that has the same time complexity as the original algorithm. On the other hand, if the output is a scalar variable, no matter how many input variables there are, *reverse-mode* automatic differentiation generates derivative code that has the same time complexity as the original algorithm. The latter case is particularly interesting, since it means that we can compute the gradient with the same time complexity (the "cheap gradient principle"), which can be useful for various optimization and sampling algorithms.

Next, we demonstrate several algorithms for computing the derivatives while carefully taking the common subexpressions into consideration. We show how to transform a numerical algorithm with control flow, loops, or recursion to code that generates the derivatives.

2.2.1 Forward-mode

We start with the simplest algorithm, usually called forward-mode automatic differentiation, and sometimes also called dual number (see Chapter 2.4 for historical remarks). Forward-mode traverses the computational graph from the inputs to outputs, computing derivatives of the intermediate nodes with respect to all input variables along the way. Forward-mode is efficient when the input dimension is low and the output dimension is high, since for each node in the computational graph, we need to compute the derivatives with respect to every single input variable.

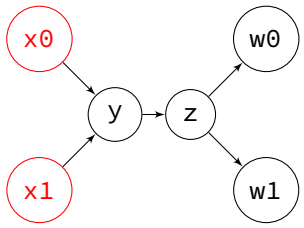
[1] However, this does not take parallelism and memory efficiency into consideration. We show in Chapter 4 how we address this issue.

In computer graphics, forward-mode has been used for computing screen-space derivatives for texture prefiltering in 3D rendering [90, 118], for computing derivatives in differential equations for physical simulation [72], and for estimating motion in specular objects [243]. Forward-mode is also useful for computing the Hessian, where one can first apply forward-mode then apply reverse-mode on each output to obtain the full Hessian matrix.

We will describe forward-mode using the previous example in Figure 2-2. Starting from the inputs, the goal is to propagate the derivatives with respect to the inputs using the chain rule. To handle function calls, for every function `f(x)` referenced by the output variables, we generate a derivative function `df(x, dx)`, where `dx` is the derivative of `x` with respect to the input variables.

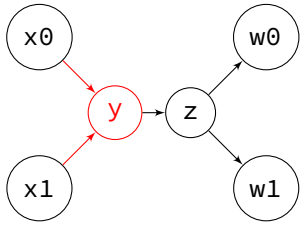
We start from the inputs `x0, x1` and generate $\frac{\partial x_0}{\partial x_0} = 1$ and $\frac{\partial x_1}{\partial x_1} = 1$. We use a 2D vector `dx0dx` to represent the derivatives of `x0` with respect to `x0` and `x1`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)
```



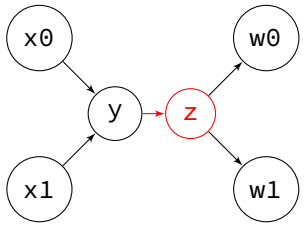
We then obtain the derivatives for `y` with respect to the inputs. We assume we already applied forward-mode automatic differentiation for `f`, so we have a derivative function `df(x0, dx0dx, x1, dx1dx)`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx,
          x1, dx1dx)
z = g(y)
w0, w1 = h(z)
```



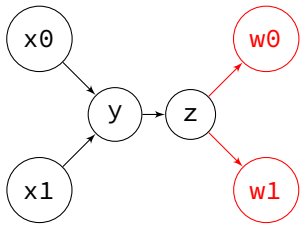
We then propagate the derivative to `z`:

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx,
          x1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = h(z)
```



Finally, we propagate the derivatives from `z` to the outputs `w0, w1`.

```
dx0dx = {1, 0}
dx1dx = {0, 1}
y = f(x0, x1)
dydx = df(x0, dx0dx,
          x1, dx1dx)
z = g(y)
dzdx = dg(y, dydx)
w0, w1 = h(z)
dw0dx, dw1dx = dh(z, dzdx)
```



The time complexity of the code generated by forward-mode automatic differentiation is $O(d)$ times the time complexity of the original algorithm, where d is the number of input variables. It is efficient for functions with few input variables.

However, for many applications of derivatives, we need to differentiate functions with thousands or even millions of input variables. Using forward-mode for this would be infeasible, as we need to compute the derivatives with respect to *all* input variables for every output in the computational graph. Fortunately, there is another algorithm called reverse-mode automatic differentiation that can generate derivative code that has the same time complexity as the original algorithm when there is only a single output, regardless of the number of input variables.

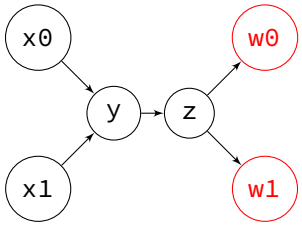
2.2.2 Reverse-mode

Reverse-mode propagates the derivatives from outputs to inputs, unlike forward-mode, which propagates the derivatives from inputs to outputs. For each node in the computational graph, we compute the derivatives of all outputs with respect to the variable at that node. Therefore reverse-mode is much more efficient when the input dimension is large and the output dimension is low. However, reverse-mode is also more complicated to implement since it needs to run the original algorithm backward to propagate the derivatives.

We again use the same previous example in Figure 2-2 to illustrate how reverse-mode works. Similar to forward-mode, we need to handle function calls. For every function `y = f(x)` referenced by the output variables, we generate a derivative function `df(x, dy)`, where dy is a vector of derivatives of the final output with respect to the function's output `y` (in contrast, in forward-mode, the derivative functions take the input derivatives as arguments). Handling control flow and recursion in reverse-mode is more complicated. We discuss them in Chapter 2.3.1.

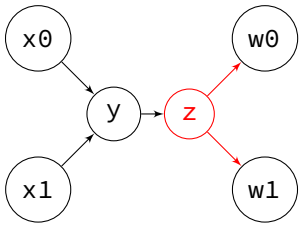
We start from the outputs `w0, w1` using $\frac{\partial w_0}{\partial w_0} = 1$ and $\frac{\partial w_1}{\partial w_1} = 1$. We use a 2D vector `dwdw0` to represent the derivatives of `w0, w1` with respect to `w0`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
```



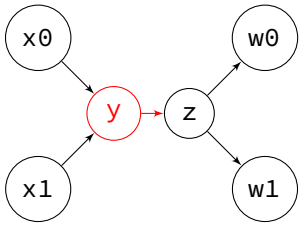
Next, we propagate the derivatives to variable `z` on which the two outputs depend. We assume we already applied reverse-mode to the function `h` and have `dh(z, dwdw0, dwdw1)`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
```



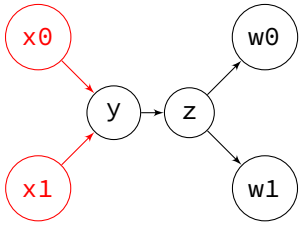
Similarly, we propagate to `y` from `z`.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
dwdy = dg(y, dwdz)
```



Finally we obtain the derivatives of the outputs `w` with respect to the two inputs.

```
y = f(x0, x1)
z = g(y)
w0, w1 = h(z)

dwdw0 = {1, 0}
dwdw1 = {0, 1}
dwdz = dh(z, dwdw0, dwdw1)
dwdy = dg(y, dwdz)
dwx0, dwx1 = df(x0, x1, dwdy)
```



A major difference between reverse-mode and forward-mode that makes the implementation of reverse-mode much more complicated, is that we can only start the differentiation after the final output is computed. This makes it impossible to interleave the derivative code with the original code like in forward-mode. This issue has the most impact when differentiating programs with control flow or recursion. We discuss them in Chapter 2.3.1.

2.2.3 Beyond Forward and Reverse Modes

As we have discussed, forward-mode is efficient when the number of inputs is small, while reverse-mode is efficient when the number of outputs is small. When both the number of inputs and the number of outputs are large, and we are interested in the Jacobian or its subset, both forward and reverse modes can be inefficient.

In general, we can think of derivative computation as a pathfinding problem on the computational graph: We want to find all the paths that connect between inputs and outputs. Many of the paths share common subpaths and it is more computationally efficient to factor out the common subpaths. Forward-mode and reverse-mode are two different greedy approaches that factor out the common subpaths either from the input node or output node, and they can deliver suboptimal results that do not have the minimal amount of computation.

For the general Jacobian, finding the factorization that results in the minimal amount of computation is called the "Jacobian accumulation problem" and is proven to be NP-Hard [154]. However, there exist several heuristics (e.g. [70, 153, 73]). Usually, the heuristics use some form of a greedy approach to factor the node that is reused by the most paths. These heuristics can also be used for higher-order derivatives such as Hessian matrices [65, 223], since the Hessian is the Jacobian of the gradient vector with respect to the input dimensions.

2.3 Automatic Differentiation as Program Transformation

In this section, we discuss the practical implementation of automatic differentiation. Typically the implementation of automatic differentiation systems can be categorized as a point in a spectrum, depending on how much is done at compile-time. At one end of the spectrum, the *tracing* approach, or sometimes called the *taping* approach, re-compiles the derivatives whenever we evaluate the function. At the other end of the spectrum, the *source transformation* approach does as much at compile-time as possible by compiling the derivative code only once. The tracing approach has the benefit of simpler implementation, and is easier to incorporate into existing code, while the source transformation approach has better performance, but usually can only handle a subset of a general-purpose language and is much more

difficult to implement.

Tracing The tracing approach bears similarity to the tracing just-in-time compilation technique used by various interpreters. Tracing automatic differentiation usually records a linear sequence of the computation at run-time (usually called a *tape* or Wengert list [227]). Typically, all the control flows will be flattened in the trace. The system then "compiles" the derivatives just-in-time by traversing the linear sequence. A typical implementation is to use operator overloading on a special floating point type, replacing all the elementary operations by the overloaded functions. The user is then required to replace all the floating point type occurences with the special type in their program, and call a compile function to start the differentiation.

Tracing is the most popular method for implementing general automatic differentiation systems. Most of the popular automatic differentiation systems use tracing (e.g. CppAD [16], ADOL-C [69], Adept [86], and Stan [207]). However, tracing is inefficient due to the limited amount of work that can be done during the just-in-time differentiation. For example, if a function is linear, all of the derivatives of it are constant, however, tracing approaches often fail to perform constant folding optimization, since folding the constant at run-time is often more costly than just computing the constant. Metaprogramming techniques such as expression templates can help mitigate this issue [86, 188], but they cannot optimize across functions or even statements.

Source Transformation Another approach is to take the source code of some numerical program, and generate the code for the derivatives. It is also possible to build an abstract syntax tree using operator overloading, then generate derivative code from the tree (the systems in Chapter 4 and Chapter 5 used precisely this approach). This approach is much more efficient compared to tracing due to the number of optimizations that can be done at compile-time (constant folding, copy elision, common subexpression elimination, etc). However, it is more difficult to integrate into existing languages, and often can only handle a subset of the language features. For example, none of the existing source transformation methods is able to handle functions with arbitrary side effects.

In Chapter 2.2 we already discussed general rules for handling elementary operations and function calls. A straightforward line to line syntax tree transformation should do the job. In the subsection below, we briefly discuss how source transformation can be done for programs with control flow including for loops and while loops, and how to handle recursion or cyclic function calls.

2.3.1 Control Flow and Recursion

Handling control flow and recursion in forward-mode is trivial. We do not need to modify the flow at all. Since forward-mode propagates from the inputs, for each statement, we can compute its derivative immediately after like we did in Chapter 2.2.1.

In reverse-mode, however, control flow and recursion introduce challenges, since we need to *revert* the flow. Consider the iterative exponential example from Figure 2-1. To apply reverse-mode, we need to revert the for loop. We observe an issue here: we need the intermediate `exp(result)` values for the derivatives. To resolve this, we will need to record the intermediate values during the first pass of the loop:

```
function d_f(x):
    result = x
    results = []
    for i = 1 to 8:
        results.push(result)
        result = exp(result)

    d_result = 1
    for i = 8 to 1:
        // one-based indexing
        d_result = d_result * exp(results[i])
    return d_result
```

The general strategy for transforming loops in reverse-mode is to push intermediate variables into a stack for each loop [217], then pop the items during the reverse loop. Nested loops can be handled in the same way. For efficient code generation, dependency analysis is often required to push only variables that will be used later to the stack (e.g. [202]).

The same strategy of storing intermediate variables in a stack also works for loop continuations, early exits, and conditioned while loops. We can use the size of the stack as the termination criteria. For example, we modify the previous example to a while loop and highlight the derivative code in red:

```
function d_f(x):
    result = x
    results = []
    while result > 0.1 and result < 10:
        results.push(result)
        result = exp(result)

    d_result = 1
    for i = len(results) to 1:
        d_result = d_result * exp(results[i])
    return d_result
```

Recursion is equally or even more troublesome compared to control flow for reverse-mode. Consider the following tail recursion that represents the same function:

```
function f(x):
    if x <= 0.1 or x >= 10:
        return x
    result = f(exp(x))
    return result
```

It is tempting to use the reverse-mode rules we developed in Chapter 2.2.2 to differentiate the function like the following:

```
function d_f(x, d_result):
    if x <= 0.1 or x >= 10:
        return 1
    result = f(exp(x))
    return d_f(result, d_result) * exp(x)
```

However, a close inspection reveals that the generated derivative function `d_f` has higher time complexity compared to the original function ($O(N^2)$ v.s. $O(N)$), since every time we call `d_f` we will recompute `f(exp(x))`, resulting in redundant computation.

A solution to this, similar to the case of loops, is to use the technique of memoization. We can cache the result of recursive function calls in a stack, and traverse the recursion tree in reverse by traversing the stack:

```
function d_f(x, d_result):
    if x <= 0.1 or x >= 10:
        return 1
    results = []
    result = f(exp(x), results)

    d_result = 1
    for i = len(results) to 1:
        d_result = d_result * exp(results[i])
    return d_result
```

This also works in the case where `f` recursively calls itself several times. A possible implementation is to use a tree instead of a stack to store the intermediate results.

The transformations above reveal an issue with the reverse-mode approach. While for scalar output, reverse-mode is efficient in time complexity, it is not efficient in memory complexity, since the memory usage depends on the number of instructions, or the length of the loops. A classical optimization to reduce memory usage is called "checkpointing". The key idea is to only push to, or to *checkpoint*, the intermediate variable stack sporadically, and recompute the loop from the closest checkpoint every time. Griewank [67] showed that by checkpointing only $O(\log(N))$ times for a loop with length $O(N)$, we can achieve memory complexity of $O(\log(N))$, and time complexity of $O(N \log(N))$ for reverse-mode.

Higher-order derivatives can be obtained by successive applications of forward- and reverse-modes. Applying reverse-mode more than once can be difficult since the stack introduces side-effects (see Chapter 2.5 for more discussions). Furthermore, in the case of the Jacobian computation, it is difficult to devise transformation rules for control flow for methods beyond forward- and reverse-modes.

2.4 Historical Remarks

Automatic differentiation is perhaps one of the most rediscovered ideas in the scientific literature. Forward-mode is equivalent to the dual number algebra introduced in 1871 [41]. The idea of reverse-mode was floating around in the 1960s (e.g. [112]), and most likely materialized first in 1970 [132] for estimating the rounding error of an algorithm, and was later applied to neural networks and rebranded as backpropagation [228, 187]. In computer graphics, the field of animation control has a long history of using automatic differentiation. Witkin and Kass developed a Lisp-based system that can automatically generate derivatives for optimizing character animation [230]. The field of optimal control theory, which is highly related to animation control, is also an early user of automatic differentiation. They take the

differential equation perspective and usually call forward-mode "tangent" or "sensitivity" while calling reverse-mode "adjoint". One of the earlier large-scale usages of automatic differentiation is oceanography (e.g. [145]), where the derivatives of fluid simulators are used for sensitivity and optimization studies. Due to the strong interest from the science and engineering communities, many early automatic differentiation tools are developed in Fortran (e.g. ADIFOR [22], TAMC [59], OpenAD [212]). See Griewank and Schmidhuber's articles [68, 191] for more remarks.

2.5 Further Readings

Deep learning frameworks The core of deep learning is backpropagation, or equivalently reverse-mode automatic differentiation. There are several recent deep learning frameworks for implementing neural network architectures. Some of them are closer to the tracing approach [165], while some of them are closer to the source transformation approach [17, 234, 1]. However, all of them only differentiate the code at a coarse-level of operators, while the operators (e.g. convolution, element-wise operations, pooling) and their gradients are implemented by experts. When the desired operation is easy to express by a few of these operators, these frameworks deliver efficient performance. However, for many novel operators, it is either inefficient or impossible to implement on top of these frameworks, and practitioners often end up implementing their own custom operators in C++ or CUDA, and derive the derivatives by hand. In Chapter 4 we discuss this in the context of image processing and deep learning.

Stochastic approximation of derivatives In addition to finite differences and symbolic differentiation, one can also employ stochastic approximation to gradients or higher-order derivatives. Simultaneous perturbation stochastic approximation (SPSA) [21] and evolution strategy [20] are two examples of this. Curvature propagation [147] takes a similar idea to stochastically approximate Hessian matrix using exact gradients. These methods sidestep the time complexity of finite differences, at the cost of having variance on the derivatives depending on the local dimensionality of the function.

Nested applications of reverse-mode An issue with the approach for handling control flow and recursion we introduced in Chapter 2.3.1 is that it does not form a *closure*, that is, the derivative code that uses the stack cannot be differentiated again, since the stack introduces side-effects. Pearlmutter and Siskind [167] propose a solution for this using Lambda calculus, by developing proper transformation rules in a side-effect free functional language, which

produces closure. The generated code has similar performance to the stack approach, but has the benefit of supporting nested applications of reverse-mode. The resulting transformation is non-local (in contrast, the one we describe in Chapter 2.2 is local), in the sense that the functions generated can be vastly different from the original ones. Recently, Shaikhha et al. [195] generalize Pearlmutter and Siskind's idea to handle array inputs in a functional language. Their current implementation does not generate vectorized code, but it is possible to further generalize their approach for better code generation.

Higher-order derivatives For Hessian computation, Gower and Mello develop a reverse-mode-like algorithm that utilizes the symmetry and sparsity [65]. It was later shown to be equivalent to one of the heursitics for computing Jacobian accumulation [223]. Betancourt [19] explores the connections between automatic differentiation and differential geometry, and develops algorithms for higher-order derivatives similar to Gower and Mello's method.

3 Derivative-based Optimization and Markov Chain Monte Carlo Sampling

Most of the uses of derivatives in this dissertation are for optimizing or sampling a function. For optimization, we are interested in the *mode* of a function, whereas for sampling we are interested in the *statistics*, such as mean or variance. In this chapter, we briefly introduce classical methods that use derivatives for optimization and Markov chain sampling. This is a massive topic and it deserves multiple university courses. Therefore, this chapter is by no means a comprehensive introduction. I only discuss methods more relevant to the dissertation. Readers are encouraged to read textbooks from Boyd and Vandenberghe [26], Nocedal and Wright [157] (both for optimization), Brooks et al. [27] (for sampling, focus on Markov chain Monte Carlo methods), and Owen [161] (for sampling, introduces various Monte Carlo integration methods).

Optimization and sampling have myriad applications across all fields of computational science. Optimization can be used for finding the parameters of a model given training input and output pairs, or solving inverse problems, where we want to find inputs that map to certain outputs. Markov chain sampling can be used for integrating light path contribution in physically-based rendering, characterizing posterior distributions in Bayesian statistics, or generating molecular structures for computational chemistry. We also discuss the relationship between optimization and sampling in Chapter 3.3.

3.1 Optimization

Given a function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, we are interested in finding an input $\mathbf{x}^*$ that minimizes the function. The function f we want to minimize is often call the *cost* function, *loss* function, or *energy* function, where the last term is borrowed from molecular dynamics. Formally, this is usually written as:

$$\mathbf{x}^* = \arg\min_{\mathbf{x}} f(\mathbf{x}). \tag{3.1}$$

For example, we may want to recover an unknown pose $\mathbf{p}$ of a camera, such that when

we pass it to a rendering function $r(\mathbf{p})$, the output matches an observation image $\mathbf{I}$. We can define the loss function as the squared difference between the rendering output and the observed image:

$$\mathbf{p}^* = \arg\min_{\mathbf{p}} \sum_i \|r(\mathbf{p}) - \mathbf{I}\|^2. \tag{3.2}$$

The goal of optimization is then to find a camera pose $\mathbf{p}$ that renders an image similar to the observation $\mathbf{I}$. These problems are usually called *inverse problems*, since we have a forward model r, and we are interested in inverting the model.

Another use case is when we have a sequence of example inputs $\mathbf{a}_i$ and outputs $\mathbf{b}_i$, and we want to learn a mapping between them. We can define the mapping as $g(\mathbf{a}_i; \Theta)$, where Θ is some set of parameters. We can then define the loss function as the difference between the mapped outputs and the example outputs:

$$\Theta^* = \arg\min_{\Theta} \sum_i \|g(\mathbf{a}_i; \Theta) - \mathbf{b}_i\|^2, \tag{3.3}$$

and optimize the mapping parameters Θ. In statistics, this is often called regression, while in machine learning this is called supervised learning, or empirical risk minimization.

Blindly searching for inputs or parameters that minimize the loss function is inefficient, especially when the dimension n is high. Intuitively speaking, the space to search grows exponentially with respect to the dimensionality. Therefore, it is important to guide the search towards a direction that lowers the cost function. This is precisely what a gradient vector does. The gradient points in the direction where the function increases the most in the infinitesimal neighborhood. If we move along the negative gradient direction, we expect the cost function to decrease. This motivates our first optimization algorithm, gradient descent.

3.1.1 Gradient Descent

The idea of gradient descent dates back to Cauchy [33]. Figure 3-1 illustrates the process. For a loss function $f(\mathbf{x})$, starting from an initial guess $\mathbf{x}_0$, we iteratively refine the guess using the gradient $\nabla f(\mathbf{x})$:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \gamma \nabla f(\mathbf{x}_i), \tag{3.4}$$

where γ is the *step size* parameter, sometimes called the *learning rate*, which determines how far we move along the negative gradient direction. Choosing the right step size is difficult, as it usually depends on the smoothness of the cost function, and typically the best step size is different for each dimension.

Gradient descent and all optimization methods we introduce in this chapter are *local*

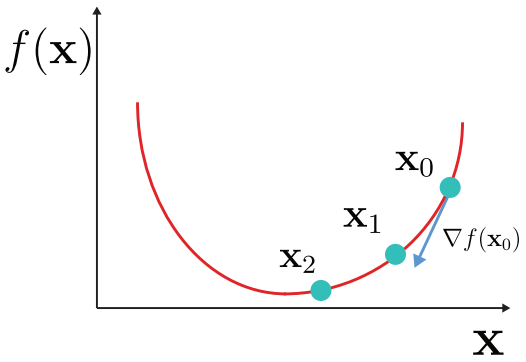


Figure 3-1: Gradient descent minimizes a function by iteratively following the negative gradient direction.

search methods. This means that they only find a local minimum of the function, while there can be a global minimum that is lower than the minimum they find.

Without any assumption on the function f, there is no guarantee that gradient descent will converge even to a local minimum. For example, if we reach a saddle point, the gradient would be zero and the iteration would stop. The convergence rate of gradient descent depends on how *convex* the function is (is it a "bowl shape" so that it only has a single minimum?), and whether it is *Lipschitz continuous* (is there a bound on how fast the function is changing?). Curious readers can consult textbooks (e.g. [26]) for more convergence proofs.

3.1.2 Stochastic Gradient Descent

In many applications, the gradient $\nabla f(\mathbf{x})$ we compute may not be fully accurate. For example, in regression, our cost function is a sum over example input-output pairs. If we have a huge database of example pairs, say one million, doing one step of gradient descent would require inefficiently enumerating all pairs of inputs and outputs. It would be desirable to randomly select a *mini-batch* each time we perform a gradient descent step (say, four from the one million). Furthermore, sometimes in an inverse problem, our forward model itself is a stochastic approximation to an integral (e.g. the rendering function in Chapter 5), and so is our loss function and gradients.

Fortunately, if our gradient approximation is *unbiased* (the expectation is the same as the true gradient) or *consistent* (the expectation converges to the true gradient if we use more samples), gradient descent can still converge to a local minimum [179, 36]. The condition for convergence is a gradually reducing step size γ over iterations, or equivalently, an increasing number of samples for gradient approximation. Intuitively, the noise we introduce in the gradient approximation brings some randomness to the steps in the gradient descent

iterations, but on average, they still go in the right direction. When we are closer to the optimum, the noise makes it harder to hit the exact optimum, so we either need to take smaller steps, or reduce the noise by increasing the number of samples.

In addition to computational efficiency, it is observed that the randomness can help stochastic gradient descent escapes from saddle points [55]. The noise also acts as an *early stopping* mechanism [170, 76], which helps regression to generalize better to data not in the examples, thereby avoiding *overfitting*. See Chapter 3.3 for more discussions on this, and the relationship between stochastic gradient descent and other sampling-based methods.

3.1.3 Newton's Method

Choosing the right step size for gradient descent methods is difficult and problem-dependent. Intuitively, for flat regions of cost functions, we want to choose a larger step size, while sharp regions require a smaller step size. Second-order derivatives are a good measure of how flat a function is: if the magnitude of the second-order derivatives is large, then the gradient is changing fast, so we should not take a large step.

In the 1D case, assuming the loss function always has positive second derivatives (which means it has a bowl shape or is convex), the update step of Newton's method is

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}, \tag{3.5}$$

where we replace the step size γ with the inverse of the second derivative.

To derive Newton's method for the multivariate case, let us expand the loss function using the second-order Taylor expansion around $\mathbf{x}_i$:

$$f(\mathbf{x}_i + \Delta_\mathbf{x}) \approx f(\mathbf{x}_i) + \nabla f(\mathbf{x}_i)\Delta_\mathbf{x} + \frac{1}{2}\Delta_\mathbf{x}^T \mathbf{H}(\mathbf{x}_i)\Delta_\mathbf{x}, \tag{3.6}$$

where $\mathbf{H}(\mathbf{x}_i)$ is the Hessian matrix. If we solve for the critical point of this approximation by taking the gradient of $\Delta_\mathbf{x}$ and setting it to zero, we arrive at an update rule:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{H}(\mathbf{x}_i)^{-1}\nabla f(\mathbf{x}_i). \tag{3.7}$$

Essentially we replace the division of the second derivative in Equation 3.5 by multiplication by the inverse of the Hessian matrix.

Newton's method can also be modified to work in a stochastic setting, where both the gradient and Hessian are an approximation to the true ones (e.g. [182, 183]).

Newton's method eliminates the need for choosing the step size, at the cost of several

disadvantages. First, the critical point of the Taylor expansion is not necessarily the minimum: it is only the minimum when the Hessian matrix is positive definite (all eigenvalues are positive). Second, computing and inverting the Hessian is expensive in high-dimensional cases. Various methods address these issues. Quasi-Newton methods or Gauss-Newton methods approximate the Hessian using first-order derivatives. Hessian-free methods (e.g. [146]) use the Hessian-vector product (much cheaper than full Hessian computation) to obtain second-order information. Some methods approximate the Hessian using its diagonal (e.g. [147]). Adaptive gradient methods adjust the learning rate per dimension using the statistics of gradients from previous iterations.

Next, we will briefly introduce adaptive gradient methods, as we use them extensively in the following chapters. We will skip the discussions on Quasi-Newton, Gauss-Newton methods and others, since they are less relevant to this dissertation.

3.1.4 Adaptive Gradient Methods

How do we assess the flatness of a function, or how fast the gradients are changing, without looking at the second-order derivatives? The idea is to look at previous gradient descent iterations. The magnitude of the gradients is often a good indicator: if the magnitude is large, the function is changing fast. *Adagrad* [49] builds on this idea and uses the inverse of average gradient magnitude per dimension as the step size:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \frac{\gamma}{\sqrt{G_i^2 + \epsilon}} \circ \nabla f(\mathbf{x}_i), \tag{3.8}$$

where G_i^2 is a vector of the *sum* of the squared gradients at or before iteration i (the *second moment* of the gradient), the division and the $\circ$ here denote element-wise division and multiplication, and ϵ is a small number (say, 10^{-8}) to prevent division by zero.

Adagrad tends to reduce the learning rate quite aggressively, since it keeps the sum of squared gradient instead of average. Also, the smoothness of a function may be significantly different during the course of optimization. A possible modification is to only keep track of recent squared gradients. This can be done by an exponential moving average update (sometimes called an infinite impulse response filter):

$$G_i'^2 = \alpha {G'_{i-1}}^2 + (1-\alpha)\nabla f(\mathbf{x}_i)^2, \tag{3.9}$$

where α is the weight update parameter. This leads us to the *RMSProp* method [208], which

replaces the second moment G^2 with the exponential moving average G'^2:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \frac{\gamma}{\sqrt{G_i'^2 + \epsilon}} \circ \nabla f(\mathbf{x}_i). \tag{3.10}$$

Finally, in the stochastic setting, the gradient can be noisy, and the exponential moving average can filter out the noise. Therefore, we can also apply the moving average to the gradient in addition to the second moment, maintaining its first moment m_i:

$$m_i = \beta m_{i-1} + (1 - \beta)\nabla f(\mathbf{x}_i), \tag{3.11}$$

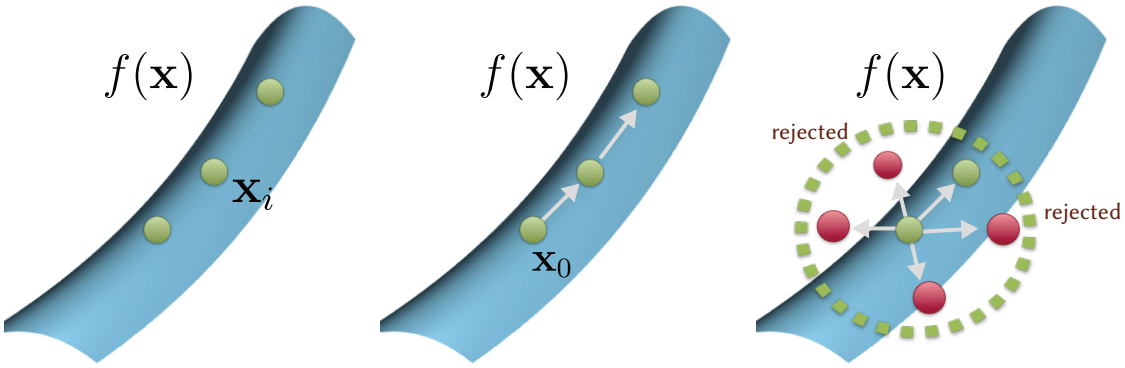
where β is another weight update parameter. We can then use this smoothed gradient for the update. This results in the most popular gradient-based optimization algorithm as of the time this dissertation is written, Adam [115][1]:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \frac{\gamma}{\sqrt{G_i'^2 + \epsilon}} \circ m_i. \tag{3.12}$$

In regression, an optimizer that achieves low error in the example pairs mapping is not necessarily going to be considered a *good* optimizer. What matters more is generalization, that is, how does the mapping perform for pairs that are not in the examples. Per discussion in Chapter 3.1.2 and 3.3, the noise in stochastic gradient descent sometimes acts as an early termination mechanism, making the loss function higher, but also making the mapping generalize better. This might also explain why most efforts on improving Adam recently are not replacing it for regression tasks. There are theories explaining the generalization behavior of stochastic gradient descent [76, 238]. However, to the author's knowledge, so far no theory explains the differences in the generalization ability between different adaptive gradient methods.

There are many other variants of adaptive gradient methods, and they are still being actively developed. ADADELTA [236] is an alternative that also keeps track of the second moment of the updates (in addition to just the gradient second moment). The moving average for the gradients in Adam is essentially the same as a popular acceleration method for gradient descent called momentum [63]. Nesterov [156] proposes an acceleration by extrapolating the momentum, achieving the same convergence rate as Newton's method in the convex and non-stochastic setting. It is also possible to incorporate Nesterov's method in Adam [47]. Reddi et al. [174] study the convergence of Adam and find counterexamples in the convex

[1] I omit the bias correction for the moving average here for simplicity. See the original paper for more details.



(a) sampling (b) Markov chain Monte Carlo (c) proposal

Figure 3-2: **Sampling.** (a) Given a function f (2D in this case), the goal of sampling is to produce a set of samples $\mathbf{x}_i$ such that their distribution is proportional to f. (b) Markov chain Monte Carlo samples from a function by generating a sequence of samples through a *local* random walk. (c) Each sample is generated from the previous one, and probabilistically rejected if the contribution is low, so that we have a higher probability of staying in high contribution regions.

setting where Adam does not converge. They propose a fix by using the maximum second moment. Maclaurin et al. [143] show that it is possible to optimize the hyperparameters of adaptive gradient descent methods by performing reverse-mode automatic differentiation on top of gradient descent. Stochastic Average Gradient [192] and Stochastic Variance Reduced Gradient [102] focus on the mini-batch setting and perform variance reduction on the gradients by reusing previous mini-batches.

3.2 Markov Chain Monte Carlo Sampling

In this section, we discuss sampling, an operation related to optimization. In contrast to optimization which is finding the *mode* of a function, we are interested in the *statistics* such as mean or variance. I focus most of the discussion on Markov chain Monte Carlo methods, since they are more related to the derivative-based scenario. For Monte Carlo integration in general, the reader can consult Owen's textbook for more information [161]. For Monte Carlo integration for light transport, Veach's thesis [213] and the textbook by Pharr et al. [168] are both excellent references.

Figure 3-2 illustrates our goal: Given a positive function $f(\mathbf{x}) : \mathbb{R}^n \to \mathbb{R}$, we want to generate a set of random samples $\mathbf{x}_i$, such that their probability densities $p(\mathbf{x}_i)$ are proportional

to f:

$$\mathbf{x}_i \sim p(\mathbf{x}_i) \tag{3.13}$$

$$p(\mathbf{x}_i) \propto f(\mathbf{x}_i). \tag{3.14}$$

This is highly related to the optimization problems: given a function g to minimize, we can set $f = e^{-g}$ and sample from f, and pick the sample with highest f. Sampling has many uses in statistics, machine learning, and computer graphics. It is useful for estimating uncertainty: for example, every sample in Figure 3-2 achieves a high score, indicating that the problem is *ill-posed*, in the sense that many inputs have an equally good loss. For both inverse problems and regression, sampling is also a more natural solution in high-dimensional space from a probabilistic viewpoint (Chapter 3.3). Finally, in physically-based rendering, we estimate the total energy passing through each pixel by sampling light paths connecting light sources to the eye.

There are many ways to sample from a function, but most of them do not generalize to arbitrary functions. Inverse transform sampling requires us to integrate f to obtain the probability density and cumulative density function, and then invert the cumulative density functions. Rejection sampling is typically inefficient in high-dimensional space and requires us to have an upper bound on f.

We focus on a specific method for sampling from a function – the Markov chain Monte Carlo method. It does not assume much on the function it samples from, and more importantly, there are extensions of Markov chain Monte Carlo methods that make use of derivatives of the function, making the sampling more directed in high-dimensional space. The downside is it generates correlated samples that reduce sampling efficiency.

3.2.1 Metropolis-Hastings Algorithm

Markov chain Monte Carlo generates samples in a sequence, forming a *Markov chain* (Figure 3-2b). That is, the generation of each sample only depends on the previous sample. This allows us to employ a local random walk strategy like the ones we used for optimization. To generate a new sample from the current one, we define a proposal distribution $Q(\mathbf{a} \to \mathbf{b})$, we then probabilistically accept or reject the proposal based on its contribution f (Figure 3-2c). Overall the algorithm generates a sequence of samples $\mathbf{x}_t$ as follows:

1. Propose a new sample $\mathbf{x}'$ from the current sample $\mathbf{x}_t$ according to the proposal distribution: $\mathbf{x}' \sim Q(\mathbf{x}_t \to \mathbf{x}')$.

2. Compute acceptance probability $a(\mathbf{x}', \mathbf{x}_t) = min\left(1, \frac{f(\mathbf{x}')}{f(\mathbf{x}_t)} \frac{Q(\mathbf{x}' \to \mathbf{x}_t)}{Q(\mathbf{x}_i \to \mathbf{x}')}\right)$

3. Set $\mathbf{x}_{t+1} = \mathbf{x}'$ if accepted, otherwise $\mathbf{x}_{t+1} = \mathbf{x}_t$.

The algorithm was developed by Metropolis [149] for symmetric proposal distributions, later extended by Hastings [78] for handling asymmetric proposals, and extended again by Green [66] for handling spaces of varying dimensions. Intuitively speaking, this algorithm allows us to put more samples in the high contribution regions, while having non zero probability of visiting all of f's domain. Below we provide a sketch of proof explaining why the sequence $\mathbf{x}_t$ is distributed proportionally to f.

It is easier to explain Markov chains in the discrete state space. Let us for now assume $\mathbf{x}_t$ represents a positive integer and f maps from $\mathbb{N}$ to $\mathbb{R}$. Our transition distribution Q becomes a matrix Q_{ij} representing the probability to transition from i to j, and the acceptance probability is also a matrix a_{ij}. All the statements below naturally generalize to continuous state space.

First, we need to define the concept of a *stationary distribution*. We represent our current sample distribution as a probability mass function vector π^t. Each iteration in the Markov chain, is essentially transforming the probability mass function:

$$K\pi^t = \pi^{t+1}, \tag{3.15}$$

where K is the *kernel* matrix of the Markov chain. We say π is a stationary distribution of the kernel K if $K\pi = \pi$. In other words, π is a fixed point of the kernel K, or π is the eigenvector corresponding to eigenvalue 1. If a Markov chain is *ergodic*, that is, after enough transitions, a state has a non-zero probability of reaching all states, then it has a unique stationary distribution. This means that, in the limit, any distribution will converge to the stationary distribution after enough iterations.

Next, we define the *detailed balance* condition. A Markov chain with kernel K and a distribution π is said to satisfy the detailed balance condition if:

$$K_{ij}\pi_i = K_{ji}\pi_j \;\forall\, i, j, \tag{3.16}$$

where the kernel matrix K_{ij} describes the probability of state i transitioning to state j. Intuitively, this means that the probability of transitioning from i to j is the same as from j to i. A kernel satisfying detailed balance implies that it has a stationary distribution, but a kernel with a stationary distribution does not necessarily satisfy detailed balance. This can be observed by summing over j in Equation 3.16:

$$\sum_j K_{ij}\pi_i = \pi_i = \sum_j K_{ji}\pi_j, \tag{3.17}$$

where the first equation comes from $\sum_j K_{ij} = 1$ since the probabilities of state transition sum to one.

Finally, we show that the kernel specified by the Metropolis-Hastings algorithm satisfies detailed balance for the distribution proportional to f. Therefore, as long as the transition distribution T is ergodic, it will converge to the right solution. We observe that state i transitions to state j with probability $T_{ij}a_{ij}$ (accept), and stays in i with $\sum_j T_{ij}(1-a_{ij})$ (reject). Hence the kernel K is:

$$K_{ij} = \begin{cases} T_{ij}a_{ij}, & \text{if } i \neq j \\ T_{ii}a_{ii} + \sum_j T_{ij}(1-a_{ij}), & \text{i = j} \end{cases}. \tag{3.18}$$

By substituting $a_{ij} = min\left(1, \frac{f_j T_{ji}}{f_i T_{ij}}\right)$ into the kernel K_{ij}, and applying some algebra, it can be shown that $K_{ij}f_i = K_{ji}f_j$.

The same proof also applies to the continuous state space by replacing all sums with integrals.

While Metropolis-Hastings generates a correct distribution in the limit, the rate it reaches that limit can vary (usually called the *mixing rate*). The success of Markov chain Monte Carlo methods depends on the transition kernels. If most proposals are rejected, we stay in the same state and waste many samples. On the other hand, even if all the proposals are accepted, if we do not move away enough from the starting position to explore the state space, we still get a bad mixing rate.

Similarly, in the optimization case, blindly moving samples around (say, using an isotropic Gaussian distribution as proposal distribution) can be inefficient, especially in the high-dimensional case and when the contribution function is sparse. Below we discuss two variants of Markov chain Monte Carlo methods that use derivatives to improve the mixing rate – Langevin Monte Carlo and Hamiltonian Monte Carlo.

3.2.2 Langevin Monte Carlo

Langevin Monte Carlo, or the Metropolis-adjusted Langevin Algorithm [181], while derived from Langevin dynamics for describing the behavior of molecules, has a pretty simple intuition: it follows the gradient flow by using a proposal distribution whose center is shifted by the gradient (Figure 3-3). Formally the transition from state $\mathbf{x}$ to state $\mathbf{y}$ is:

$$T(\mathbf{x} \to \mathbf{y}) \sim \mathcal{N}(-\gamma \nabla f(\mathbf{x}), \sigma^2 I), \tag{3.19}$$

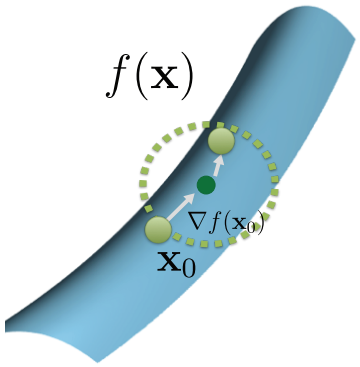


Figure 3-3: Langevin Monte Carlo, or the Metropolis-adjusted Langevin Algorithm [181] follows the gradient field by generating proposals from an isotropic Gaussian distribution, whose mean is shifted by the gradient of the sampling function.

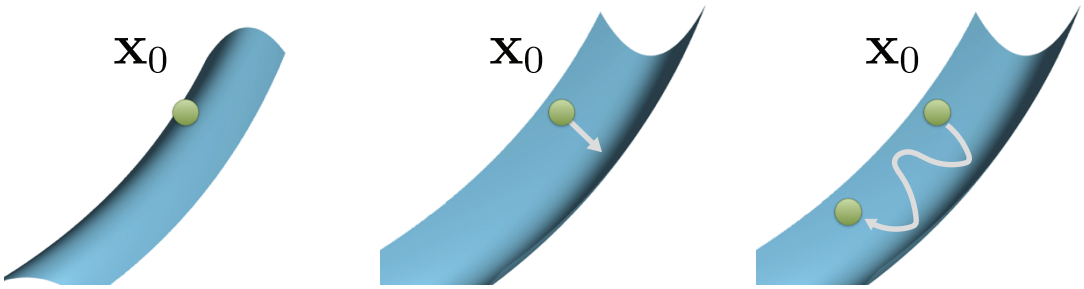
where $\mathcal{N}$ is the normal distribution, γ is the step size or learning rate, σ is a scalar standard deviation, and I is the identity matrix.

This simple modification already brings significant benefits. Roberts and Rosenthal [180] show that if $f(\mathbf{x})$ is high-dimensional (say, more than 5) and *separable* (the dimensions of the input $\mathbf{x}$ are independent to each other), then the *optimal* acceptance rate of Langevin Monte Carlo is around 57%, while the optimal acceptance rate of the Metropolis algorithm using isotropic Gaussian is around 23%. This means that the sampling efficiency of Langevin Monte Carlo is much better than zero-mean isotropic Gaussian in this case. Langevin Monte Carlo also produces less correlated samples. For d-dimensional separable functions, the expected number of samples needed to reach a nearly independent point grows as $d^{\frac{4}{3}}$, where when using isotropic Gaussian the number grows as d^2 [155].

3.2.3 Hamiltonian Monte Carlo

Hamiltonian Monte Carlo [48] takes the idea of following gradient flow further, blurring the distinction between sampling and optimization. Figure 3-4 illustrates a mental model for Hamiltonian Monte Carlo: we first flip the landscape of f upside down, moving high contribution regions to the lower ground. Assuming the current sample is a rigid ball, we assign a random initial velocity to the ball, and simulate physics to let gravity pull the ball towards the lower ground, which are the higher contribution regions since we flipped f. In Chapter 6 we build on this idea to develop a Markov chain Monte Carlo rendering algorithm. We will provide a more formal introduction of Hamiltonian Monte Carlo and related work there.

The result of this physics simulation is that we follow the gradient field of the function guided by a momentum (similar to the gradient descent momentum we discussed in Chap-



(a) original landscape (b) flip, random velocity (c) simulate dynamics

Figure 3-4: Hamiltonian Monte Carlo [48] takes the sampling function (a) and flips it upside down (b). It then assigns a random velocity to the sample and simulates physics to let gravity pull the balls and stop at some fixed time (c).

ter 3.1.4). Comparing to Langevin Monte Carlo, Hamiltonian Monte Carlo reduces the randomness by only accepting or rejecting a sample after a fixed time. It scales even better with dimensionality ($d^{\frac{5}{4}}$ for d-dimensional separable functions) and also has a better optimal acceptance rate (65% for high-dimensional separable functions). The downside is it needs to discretize the physics simulation into multiple timesteps, making the cost of generating a sample very high. In fact, Langevin Monte Carlo is a special case of Hamiltonian Monte Carlo with only a single time step. The method we develop in Chapter 6 combines the benefits of Hamiltonian Monte Carlo and Lanvegin Monte Carlo by simulating Hamiltonian dynamics in a local neighborhood, while potentially sacrificing some benefits of the reduced randomness.

3.2.4 Stochastic Langevin or Hamiltonian Monte Carlo

Gradient descent works when the gradient is a stochastic approximation. Fortunately, Langevin Monte Carlo and Hamiltonian Monte Carlo also work when gradients are stochastic [155, 226, 38]. It may seem trivial, since any proposal distribution that satisfies detailed balance should converge (the gradients do not even need to be unbiased or consistent). However, especially in the case of Hamiltonian Monte Carlo, to ensure a good convergence rate, care has to be taken to balance the dynamics to counter the noise injected in the trajectory [38].

3.3 Relation between Optimization and Sampling

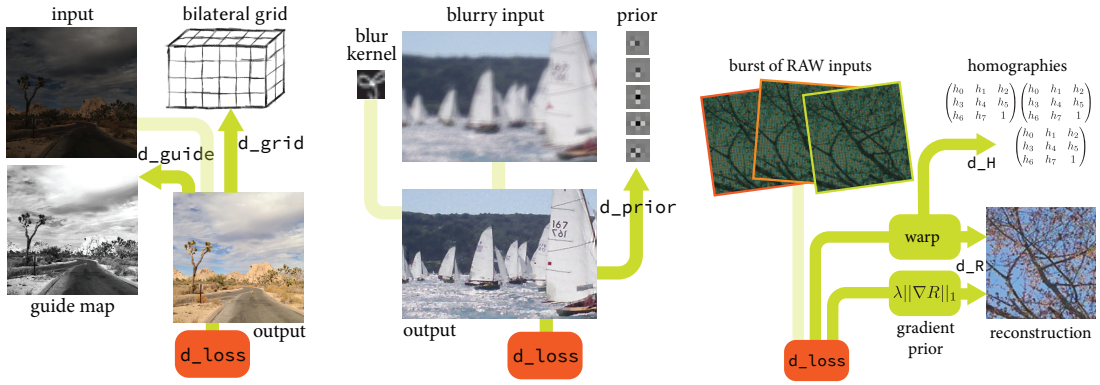
From a probabilistic viewpoint, we can view optimization as trying to find a point that is maximizing the probability density distribution. For example, if we use a squared loss $f(\mathbf{x}) = |g(\mathbf{x}) - \mathbf{y}|^2$, we can see this as finding the mode of a normal distribution centered around $\mathbf{y}$ (more precisely, the density is $p(g(\mathbf{x})) \propto e^{\frac{-f(\mathbf{x})}{\sigma^2}}$ for some standard deviation

σ representing the uncertainty). If we use the absolute difference, often called L^1 loss, it corresponds to the Laplace distribution. In contrast to optimization, sampling algorithms try to *sample* from the Gaussian distribution, so points with higher density are more likely to be sampled.

Recently, researchers have started to explore the relationship between Markov chain Monte Carlo sampling and various gradient descent methods. In certain non-convex settings for optimization, Markov chain Monte Carlo methods, when used for optimization, have a faster convergence rate than gradient-based optimization algorithms [141]. On the other hand, some variants of Langevin Monte Carlo always accept proposals [226], making the methods resemble gradient-based optimization more. There are many similar parallel developments between the sampling and optimization literature. Hamiltonian Monte Carlo's introduction of momentum to Langevin Monte Carlo is similar to the momentum in gradient descent. Gibbs' sampling [56] is similar to coordinate descent by treating only a subset of input variables at a time. Riemannian Manifold Langevin and Hamiltonian Monte Carlo [60] introduces the second-order derivatives similar to Newton's method or natural gradient method [6]. It is fair to expect sampling and optimization algorithms to converge in the future, making it unnecessary to distinguish between them.

The connection between sampling and the generalizing effect of stochastic gradient descent [76, 144] is also worth noting. In a high-dimensional space, most of the mass of distribution does not distribute around its mode [32]. The intuition is that it is very unlikely that a person has the average height, average weight, average size of eyes and mouth, average length of arms and legs. Therefore, probabilistically, it makes little sense to find the exact minimum in high-dimensional space, since the minimum is not representative of the distribution. When we sample from a high-dimensional distribution, most of the samples would not be around the mode, but have a small distance to it. This is exactly what the noise in stochastic gradient descent is doing: in a practical number of iterations, it makes the optimization miss the exact minimum, but end up in a position having a small distance to the minimum. In effect this allows stochastic gradient descent to achieve better generalization, since they find a more typical instance of the probability distribution.

4 | Differentiable Image Processing and Deep Learning in Halide



(a) neural network operator: bilateral slicing

(b) optimizing the parameters of a *forward* image processing pipeline

(c) optimizing the reconstruction and warping parameters of an *inverse* problem

Figure 4-1: **Differentiable image processing.** Our system automatically derives and optimizes gradient code for general image processing pipelines, and yields state-of-the-art performance on both CPUs and GPUs. This enables a variety of imaging applications, from training custom neural network layers (a), to optimizing the parameters of traditional image processing pipelines (b), to solving inverse problems (c). To support these applications, we extend the Halide language to automatically and efficiently compute gradients. We also introduce a new automatic performance optimization that can handle the specific computation patterns of the gradient. Using our system, a user can easily write high-level image processing algorithms, and then automatically derive high-performance gradient code for CPUs, GPUs, and other architectures. Images from left to right are from MIT5k dataset [30], ImageNet [45], and deep demosaicking dataset [57], respectively.

Optimization and end-to-end learning are driving rapid progress in graphics and imaging, by viewing either the output image or large sets of pipeline parameters as unknowns, e.g. [80, 94, 13, 58]. Key to this progress is the surprising power of gradient-based optimization methods to find solutions to nonlinear objectives over large sets of unknowns. Unfortunately, the computation of gradients remains a challenge in the general case, especially when performance is paramount such as for training neural networks or when solving for images via optimization. In Chapter 2, we discussed methods for generating derivative code from programs, but they are not designed for image processing programs, and they do not take

parallelism and locality into consideration. Typically, practitioners have to either manually derive gradients or they are limited to the composition of building blocks offered by deep learning libraries. The result is often inefficient, and when users decide to stray from existing operators, the implementation of fast GPU derivative code is a major undertaking.

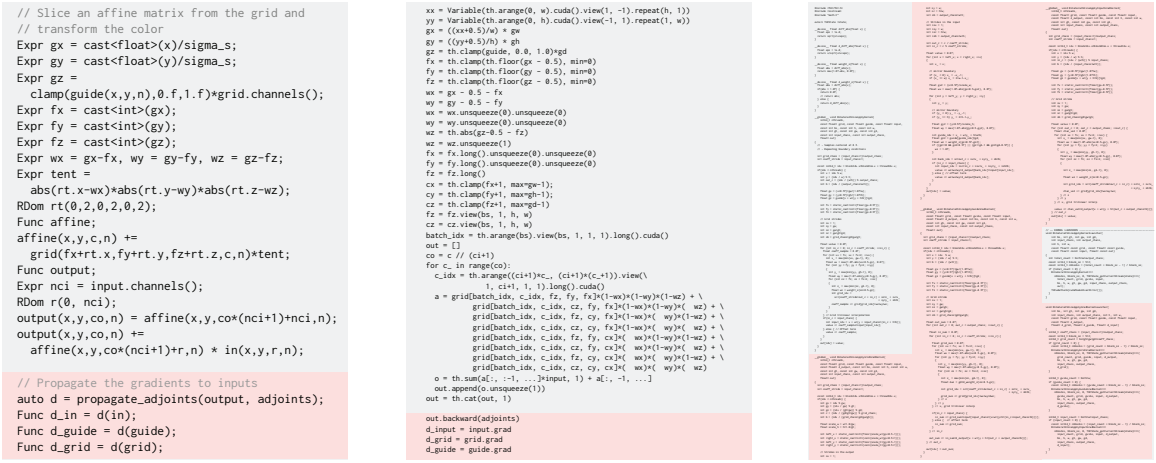
At first glance, modern machine learning frameworks like PyTorch [165], TensorFlow [1] or CNTK [234] seem like appealing environments for new gradient-based graphics algorithms. When limited to their walled-gardens of pre-made, coarse-grained operations, these frameworks provide high-performance kernel implementations and automatic differentiation through chains of operations. As general programming languages, however, they are a poor fit for many imaging applications. Building new algorithms requires contorting a problem into complex and tangled compositions of existing building blocks. Even when done successfully, the resulting implementation is often both slow and memory-inefficient, saving and reloading entire arrays of intermediate results between each step, causing costly cache misses.

Consider the following example. A recent neural network-based operator for approximating image processing algorithms was built around a new "bilateral slicing" layer based on the bilateral grid [58, 35]. At the time it was published, neither PyTorch nor TensorFlow was even capable of practically expressing this computation.[1] As a result, the authors had to define an entirely new operator, written by hand in about 100 lines of CUDA for the forward pass and 200 lines more for its manually-derived gradient (Figure 4-2, right). This was a sizeable programming task which took significant time and expertise. While new operations now make it possible to implement this operation in 42 lines of PyTorch, this yields less than 1/3rd the performance on small inputs and runs out of memory on realistically-sized images (Figure 4-2, middle). The challenge of efficiently deriving and computing gradients for custom nodes remains a serious obstacle to deep learning.

This pattern is ubiquitous. New custom nodes require major effort to implement correctly and efficiently, making it hard to experiment. Similarly, general image processing pipelines often do not map well to deep learning toolboxes. As a result, most researchers limit themselves to consider only operations which are already well-supported by existing frameworks, while NVIDIA and the framework developers must constantly expand the set of native operations. The only alternative is to invest orders of magnitude more effort in developing custom operations, hand-deriving, reimplementing, and debugging gradient code for every change during the development of a new algorithm.

Recently, the Halide domain-specific language [171, 172] has enabled the implementation

[1] Technically, TensorFlow graphs are Turing-complete, thanks to their inclusion of a while loop node. However, implementing the algorithm at this level would be both incredibly complex and run at least thousands of times slower.



```
// Slice an affine matrix from the grid and
// transform the color
Expr gx = cast<float>(x)/sigma_s;
Expr gy = cast<float>(y)/sigma_s;
Expr gz =
  clamp(guide(x,y,n),0.f,1.f)*grid.channels();
Expr fx = cast<int>(gx);
Expr fy = cast<int>(gy);
Expr fz = cast<int>(gz);
Expr wx = gx-fx, wy = gy-fy, wz = gz-fz;
Expr tent =
  abs(rt.x-wx)*abs(rt.y-wy)*abs(rt.z-wz);
RDom rt(0,2,0,2,0,2);
Func affine;
affine(x,y,c,n) +=
  grid(fx+rt.x,fy+rt.y,fz+rt.z,c,n)*tent;
Func output;
Expr nci = input.channels();
RDom r(0, nci);
output(x,y,co,n) = affine(x,y,co*(nci+1)+nci,n);
output(x,y,co,n) +=
  affine(x,y,co*(nci+1)+r,n) * in(x,y,r,n);

// Propagate the gradients to inputs
auto d = propagate_adjoints(output, adjoints);
Func d_in = d(in);
Func d_guide = d(guide);
Func d_grid = d(grid);
```

Halide Runtime
24 lines 64 ms (1 MPix)
165 ms (4 MPix)

```
xx = Variable(th.arange(0, w).cuda().view(1, -1).repeat(h, 1))
yy = Variable(th.arange(0, h).cuda().view(-1, 1).repeat(1, w))
gx = ((xx+0.5)/w) * gw
gy = ((yy+0.5)/h) * gh
gz = th.clamp(guide, 0.0, 1.0)*gd
fx = th.clamp(th.floor(gx - 0.5), min=0)
fy = th.clamp(th.floor(gy - 0.5), min=0)
fz = th.clamp(th.floor(gz - 0.5), min=0)
wx = gx - 0.5 - fx
wy = gy - 0.5 - fy
wx = wx.unsqueeze(0).unsqueeze(0)
wy = wy.unsqueeze(0).unsqueeze(0)
wz = th.abs(gz-0.5 - fz)
wz = wz.unsqueeze(1)
fx = fx.long().unsqueeze(0).unsqueeze(0)
fy = fy.long().unsqueeze(0).unsqueeze(0)
fz = fz.long()
cx = th.clamp(fx+1, max=gw-1);
cy = th.clamp(fy+1, max=gh-1);
cz = th.clamp(fz+1, max=gd-1)
fx = fz.view(bs, 1, h, w)
cz = cz.view(bs, 1, h, w)
batch_idx = th.arange(bs).view(bs, 1, 1, 1).long().cuda()
out = []
co = c // (ci+1)
for c_ in range(co):
  c_idx = th.arange((ci+1)*c_, (ci+1)*(c_+1)).view(\
              1, ci+1, 1, 1).long().cuda()
  a = grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, fz, fy, fx]*(1-wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, fz, cy, fx]*(1-wx)*(  wy)*(  wz) + \
          grid[batch_idx, c_idx, cz, fy, fx]*(  wx)*(1-wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, fy, cx]*(  wx)*(1-wy)*(  wz) + \
          grid[batch_idx, c_idx, cz, cy, fx]*(  wx)*(  wy)*(1-wz) + \
          grid[batch_idx, c_idx, cz, cy, cx]*(  wx)*(  wy)*(  wz)
  o = th.sum(a[:, :-1, ...]*input, 1) + a[:, -1, ...]
  out.append(o.unsqueeze(1))
out = th.cat(out, 1)

out.backward(adjoints)
d_input = input.grad
d_grid = grid.grad
d_guide = guide.grad
```

PyTorch Runtime
42 lines 1440 ms (1 MPix)
out of memory (4 MPix)

CUDA Runtime
308 lines 430 ms (1 MPix)
2270 ms (4 MPix)

Figure 4-2: **Code comparison.** Implementations of the forward and gradient computations of the bilateral slicing layer [58] in Halide, PyTorch, and CUDA. Using our automatic differentiation and scheduling extensions, the Halide implementation is clear, concise, and fast. The PyTorch implementation is modestly more complex, but runs 20× slower on a $1k \times 1k$ input, fails to complete (out of memory on a 12GB NVIDIA Titan Xp) on a $2k \times 2k$ input, and is only possible thanks to new operators added to PyTorch since the original publication. The CUDA implementation, developed by the original authors, is not only complex (an order of magnitude larger than either Halide or PyTorch), but is dominated by hand-derived gradient computations. It is faster than PyTorch and scales to larger inputs, but is still about 10× slower than the Halide version. Note: code size includes a few lines beyond the core logic shown for both Halide and PyTorch.

of high-performance image-processing pipelines. It is an effective solution to implementing custom nodes and general image processing pipelines, but it still requires the manual derivation of gradients. Furthermore, our experience shows that the computation pattern of derivatives differs from that of forward code, which causes existing automatic performance optimizations in Halide to fail. Critically, the current built-in Halide autoscheduler does not support GPU schedules.

We extend Halide with methods to automatically and efficiently compute the gradients of arbitrary Halide programs using reverse-mode automatic differentiation (Chapter 4.3). This transformation supports most existing features in the language, except for a few cases where side-effects are introduced (Section 4.3.2).

Building atop Halide has several advantages. It provides a concise, natural language in which to express image processing computations, and for which there is already a library of existing algorithms. The Halide compiler portably targets numerous processor and accelerator architectures, from mobile CPUs, to image processing DSPs, to data center GPUs, and supports compilation to very high-performance code. Finally, Halide's existing language and

scheduling constructs compose with reverse-mode automatic differentiation to naturally express and generalize essential optimizations from the traditional automatic differentiation literature (Chapter 4.3.3). Keys to making our compiler transformation work are a scatter-to-gather conversion algorithm which preserves parallelism (Chapter 4.3.2), and a simple automatic scheduling algorithm specialized to the patterns that appear in generated gradient code (Chapter 4.3.4). Halide's existing system of powerful dependence analyses is essential for both. In contrast to traditional Halide, automatic scheduling is critical given the complexity of the automatically-generated gradient code.

Using our new automatic gradient computation and automatic scheduler, we show how we can easily implement three recently-proposed neural network layers using code that is both faster and significantly simpler than the authors' original custom nodes written in C++ and CUDA (Chapter 4.4.1). For example, the aforementioned bilateral slicing layer is expressed in 24 lines of Halide (Figure 4-2, left), including just four lines to compute and extract its gradients, while compiling automatically to an implementation about 10× faster than the authors' original handwritten CUDA, and 20× faster than a more limited version in PyTorch. We believe that this ease of implementation and performance tuning will dramatically facilitate prototyping, by delivering both automatic gradients and high performance at the outset of experimentation, not after-the-fact once the usefulness of a node has been established.

We also argue that this approach of gradient-based optimization through arbitrary programs is useful outside the traditional deep learning applications which have popularized it. Our vision is that any image-processing pipelines can benefit from an automatic tuning of internal parameters. Currently, this step is usually done by hand through user trial-and-error. The availability of automatic derivatives makes it possible to systematically optimize any internal parameter of an image processing pipeline, given some output objectives. This is especially appealing when gradients are available in the same language used for high-performance code deployment. We show how to significantly improve the performance of two traditional image processing algorithms by automatically optimizing their key parameters and filters (Chapter 4.4.2). We also develop a novel joint burst demosaicking and superresolution algorithm by inverting a forward image formation model including warps by unknown homographies, solving for the image and homographies simultaneously (Chapter 4.4.3). Finally, we show the versatility of our approach and implement a lens design optimization by differentiating an optical simulator and fluid simulator (Chapter 4.4.4).

4.1 Related Work

4.1.1 Automatic Differentiation and Deep Learning Frameworks

Following the methods in Chapter 2, it is possible to generate derivative code from a given program. Many automatic differentiation frameworks have been developed for general programming languages [22, 69, 77, 86, 229], but general programming languages can be cumbersome for image processing applications. Writing efficient image processing code requires enormous effort to take parallelism, locality, and memory consumption/bandwidth into account [171]. These difficulties are compounded when we also want to compute derivatives. In particular, none of the existing automatic differentiation compilers or libraries can handle automatic differentiation of vectorized code.

Recent deep learning packages provide higher level, highly optimized differentiable building blocks for users to assemble their program [17, 234, 1, 165]. These packages are efficient when the algorithm to be implemented can be conveniently expressed by combining these building blocks. But it is quite common for users to write their own custom operators in low-level C++ or CUDA to extend a package's functionalities.

Using our approach, one can simply write the forward program. Our algorithm generates the derivatives and, thanks to Halide's decoupling of algorithm and schedule and our automatic scheduler, provides convenient handles to easily produce efficient code.

4.1.2 Image Processing Languages

Our work builds on the Halide [171] image processing language, which we briefly introduce in Chapter 4.2.

The Opt language [46] focuses on nonlinear least squares problems. It provides language constructs to describe the least squares cost and automatically generates solvers. It uses the D* algorithm [73] to generate derivatives for the Jacobian. The ProxImaL [79] language, on the other hand, focuses on solving inverse problems using proximal gradient algorithms. The language provides a set of functions and their corresponding proximal operators. It then generates Halide code for optimization. Our system can be used to generate the adjoints required by new ProxImaL operators.

These languages focus on a specific set of solvers, namely nonlinear least squares and proximal methods, and provide high-level interfaces to them. On the other hand, we deal with any problem that requires the gradient of a program. Our system can also be used to solve for unknowns other than images, such as optimizing the hyperparameters of an algorithm or jointly optimizing images and parameters. Chapter 4.4.3 demonstrates this with

some examples.

Recently, there have been attempts to automatically speed-up image processing pipelines [152, 233, 151, 198, 11]. We developed a new automatic scheduler in Halide with specialized mechanisms for parallel reductions [203], which often occur in the gradients of image processing code. Our system could further benefit from future developments in automatic code optimization.

4.1.3 Learning and Optimizing with Images

Gradient-based optimization is commonly used in image processing. It has been used for image restoration [186], image registration [244], optical flow estimation [89], stereo vision [13], learning image priors [184, 211] and solving complex inverse problems [80]. Our work alleviates the need to manually derive the gradient in such applications, which enables faster experimentation.

Deep learning has revitalized interest in building differentiable forward image processing pipelines whose parameters can be tuned by stochastic gradient descent. Successful instances include image restoration [57, 239], photographic enhancement [231], and applications such as colorization [91, 240], and style transfer [54, 140]. Some of these methods call for custom operators [94, 92, 58], typically not available in mainstream frameworks. For these custom operators, forward and gradient operations are implemented manually. Our work provides a convenient way to explore new custom computations.

4.2 The Halide Programming Language

Our system extends the Halide programming language. We give a brief overview of the constructs in Halide that are relevant to our system. For more detail on Halide, see the original papers [171, 172] and documentation.[2]

Halide is a language designed to make it easy to write high-performance image- and array-processing code. The key idea in Halide is the separation of a program into the *algorithm*, which specifies *what* is computed, and the *schedule*, which dictates the *order* of computation and storage. The algorithm is expressed as a pure functional, feed-forward pipeline of arithmetic operations on multidimensional grids. The *schedule* addresses concerns such as tiling, vectorization, parallelization, mapping to a GPU, etc. The language guarantees that the output of a program depends only on the *algorithm* and not on the *schedule*. This frees the user from worrying about low-level optimizations while writing the high-level algorithm.

[2] `http://halide-lang.org/`

They can then explore optimization strategies without unintentionally altering the output.

By adding automatic differentiation to Halide, we build on this philosophy. To create a differentiable pipeline, the user no longer needs to worry about the correctness and efficiency of the gradient code. With the sole specification of a forward algorithm, our system synthesizes the gradient algorithm. Optimization strategies can then be explored for both, either manually or with an auto-scheduler.

The following code shows an example Halide program that performs gamma correction on an image and computes the L^2 norm between the output and a target image:

```
Param<float> g; // Gamma parameter
Buffer<float> im, tgt; // 2-D input and target buffers
Var x, y; // Integer variables for the pixel coordinates
Func f; // Halide function declarations
// Halide function definition
f(x, y) = pow(im(x, y), g);
// Reduction variables to loop over target's domain
RDom r(tgt);
Func loss; // We compute the MSE loss between f and tgt
loss() = 0.f; // Initialize the sum to 0
Expr diff = f(r.x, r.y) - tgt(r.x, r.y);
loss() += diff * diff; // Update definition
```

Halide is embedded in C++. Halide pipeline stages are called *functions* and represented in code by the C++ class `Func`. Each Halide function is defined over an n-dimensional grid. The definition of a function comprises:

- an *initial value* that specifies a value for each grid point.
- optional *recursive updates* that modify these values in-place.

The function definitions are specified as Halide *expressions* (objects of type `Expr`). Halide expressions are side-effect-free, including arithmetic, logical expressions, conditionals, and calls to other Halide functions, input buffers, or external code (such as `sin` or `exp`).

Reduction operators, such as summation or general convolution, are implemented through recursive updates of a Halide function. The domain of a reduction is represented in code as an `RDom`, which implies a loop over that domain. All loops in Halide are implicit, whether over the domain of a function or a reduction.

Scheduling is expressed through methods exposed on `Func`. There are many scheduling operators, which transform the computation to trade off between memory bandwidth, parallelism, and redundant computation. Halide lowers the schedule and algorithm into a set of loop nests and kernels. These are then compiled to machine code for various architectures. We use the CUDA and x86 backends for the applications demonstrated in this chapter.

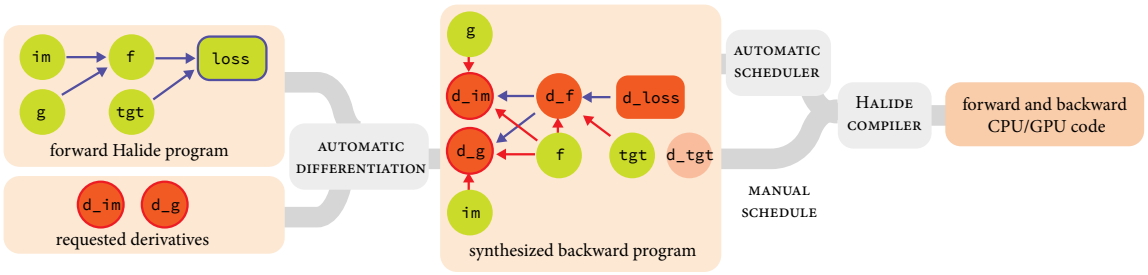


Figure 4-3: **Overview of our compiler.** The user writes a forward Halide program as they would normally. Then, they specify the set of outputs and gradients the system should produce. Our automatic differentiation generates new Halide functions that implement the requested gradients. The user can either manually schedule the pipeline or use our automatic scheduler. Finally, the Halide compiler generates machine code for the scheduled forward and backward algorithms.

4.3 Method

To use our system, a programmer first writes a forward Halide algorithm. They then request the gradient of some scalar loss with respect to any Halide function, image buffer, or parameter in the pipeline. Our automatic differentiation system visits the graph of functions that describes the forward algorithm and synthesizes new Halide functions that implement the gradient computation (Chapter 4.3.1). The programmer can either specify the schedule for these new functions manually or use our automatic scheduler (Chapter 4.3.4). Unlike Halide's built-in auto-scheduler [151], ours recognizes patterns that arise when reversing the computation graph (Chapter 4.3.2). Figure 4-3 illustrates the workflow.

4.3.1 High-level Strategy

We assume we want to compute the derivatives of some scalar $\mathcal{L}$, typically a cost function to be minimized. Our system implements reverse-mode automatic differentiation, which computes the gradient with the same time complexity as the forward function (Chapter 2.2.2). We propagate the adjoints $\frac{\partial \mathcal{L}}{\partial g}$ to each function in the forward pipeline g, until we reach the inputs. The adjoints of the inputs are the components of the gradient.

Specifically, given a Halide program represented as a graph of Halide functions, we traverse the graph backward from the output and accumulate contributions to the adjoints using the chain rule. Halide function definitions are represented as expression trees, so within each function, we perform a similar backpropagation through the expression tree, propagating adjoints to all leaves.

A key difference between our algorithm and traditional automatic differentiation arises when an expression is a Halide function call. We need to construct a computation which

accumulates adjoints onto the called function in the face of non-trivial data dependencies between the two functions. Chapter 4.3.2 describes this in detail.

We illustrate our algorithm on the example in Chapter 4.2, which performs gamma correction on an image and computes the L^2 distance between the output and some target image. To compute the gradients of the distance with respect to the input image and the gamma parameter, one would write:

```
// Obtain gradients with respect to image and gamma parameters
auto d_loss_d = propagate_adjoints(loss);
Func d_loss_d_g = d_loss_d(g);
Func d_loss_d_im = d_loss_d(im);
```

Throughout this chapter, we use the convention that prefixing a function's name with `d_` refers to the gradient of that Halide function. We extend Halide with a key feature `propagate_adjoints`. It takes a scalar Halide function and generates gradients in the form of new Halide functions for every Halide function, buffer, and real number parameter the output depends on. Our system can also be used as a component in other automatic differentiation systems that compute gradients. In this case, the user can specify a non-scalar Halide function and a buffer representing the adjoints of the function. Figure 4-3 shows the computational graph for both the original and gradient computations.

4.3.2 Differentiating Halide Function Calls

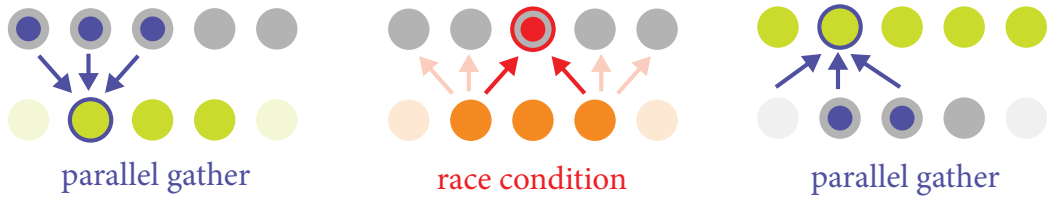
An important difference between automatic differentiation in Halide and traditional automatic differentiation, is that Halide functions are defined on multi-dimensional grids, therefore function calls and the elements on the grids can have non-trivial aggregate interactions.

Given each input-output pair of Halide functions, we synthesize a new Halide function definition that accumulates the adjoint of the output function onto the adjoint of the input. For performance, we want these new definitions to be as parallelizable as possible.

Scatter-gather conversion

Two cases require special care for correctness and efficiency. The first and most important case occurs when each output element reads and combines multiple input values. This happens for example in the simple convolution of Figure 4-4(a). We call this pattern a *gather* operation.

When computing gradients in reverse automatic differentiation, the natural reverse of this gather is a *scatter* operation: each input writes to multiple elements of the output. Scattering operations, however, are not naturally parallelizable since they may lead to race conditions on write. For this reason, we want to convert scatters back to gathers whenever possible. We do



(a) forward 1D convolution (b) backward general scatter (c) backward with our gather conversion

Figure 4-4: **Scatter-to-gather conversion.** Our compiler transform enables efficient, parallel code. In this example of a 1D 3-tap convolution, each dot represents a value in the input (resp. output) array. The forward computation (a) produces an output value from three inputs (the faded dots account for boundary conditions). This 3-tap reduction can easily be run in parallel over the output buffer (green dots). Computing the adjoint operator by simply reversing the dependency graph (b), that is by looping in parallel over the output nodes (orange), leads to race conditions since two inputs might need to write to the same location in the input's adjoint buffer (highlighted in red). This is a common issue with general scattering operations. Using our scatter-to-gather conversion, we convert this backward operation to a reduction over `d_out` (the adjoint of a convolution is a correlation). In turn, this transformed computation is readily parallelized over `d_out`'s domain (c).

this by shearing the iteration domain (e.g. [124]). To illustrate this transformation, consider the following code that convolves a 1D signal with a kernel, also illustrated in Figure 4-4(a):

```
Func output;
output(x) = input(x - r.x) * kernel(r.x);
```

Assume that we are interested in propagating the gradient to `input`. This is achieved by reversing the dependency graph between the input and output variables as shown in Figure 4-4(b). In code, this transformation would yield:

```
RDom ro;
d_input(ro.y - ro.x) += d_output(ro.y) * kernel(ro.x);
```

where `ro.x` iterates over the original `r.x`, and `ro.y` iterates over the domain of `output`. For each argument in the calls to `input`, we replace the pure variables (`x` here) with reduction variables that iterate over the domain of the output (in this case `ro.y`). `r.x` is renamed to `ro.x` so we can merge the reduction variables into a single reduction domain `ro`.

This new update definition cannot be computed in parallel over `ro.y` since multiple `ro.y - ro.x` may write to the same memory location. A more efficient way to compute the update, illustrated in Figure 4-4(c), is to rewrite the same computation as follows:

```
d_output(x) = select(x >= a && x < b, d_output(x), 0.f);
d_input(x) += d_output(x + r.x) * kernel(r.x);
```

where `a` and `b` are the bounds of `output`. By shearing the iteration domain with the variable substitution `x = ro.y - ro.x`, we have made `d_input` parallelizable over `x`. Because Halide

Listing 1 Derivatives generated by our algorithm for the bilateral slicing code in the left of Figure 4-2.

```
// We start with d_output, which contains the adjoint of output
// We propagate the derivatives from d_output to in and affine:
RDom ri(0, nci, 0, adjoints.channels());
d_in(x, y, ri.x, n) +=
  d_output(x, y, ri.y, n) * affine(x, y, ri.y * (nci + 1), n);
d_affine(x, y, ri.y*(nci+1)+ri.x, n) +=
  d_output(x, y, ri.y, n) * in(x, y, ri.x, n);
// Variable co is converted into a reduction variable rco.
RDom rco(0, adjoints.channels());
d_affine(x, y, rco*(nci+1)+nci, n) += d_output(x, y, rco, n);

// The derivatives are then propagated from affine to grid.
RDom rg(0, 2, 0, 2, 0, 2, 0, sigma_s, 0, sigma_s);
Expr inv_x = (x - rg[0]) * sigma_s + rg[3];
Expr inv_y = (y - rg[1]) * sigma_s + rg[4];
d_grid(x, y, fx + rg[2], c) +=
  d_affine(inv_x, inv_y, c, n) * d_tent;
// d_tent is tent with (x, y) replaced by (inv_x, inv_y).
// The scattering operation is transformed by solving
// x == inv_x/sigma_s+rt.x and y == inv_y/sigma_s+rt.y
// for inv_x and inv_y.

// Finally, and less obviously, affine also depends on guide.
RDom rgu(0, 2, 0, 2, 0, 2, adjoints.channels());
Expr wxy = abs(rgu[0] - wx) * abs(rgu[1] - wy);
Expr wz = select(rgu[2] - wz > 0.f, 1.f, -1.f);
d_guide(x, y, n) +=
  select(guide(x, y, n) >= 0.f && guide(x, y, n) <= 1.f,
    d_affine(x, y, rgu[3], c, n)*wxy*wz*grid.channels(), 0.f);
```

only iterates over rectangles, and the sheared iteration domain is no longer a rectangle, we add a zero-padding boundary condition to `d_output`, and iterate over a conservative bounding box of the sheared domain:

We use Halide's equation-solving tools to deduce the variable substitution to apply. For each argument in a function call, we construct an equation e.g. $u = x - r_x$ and solve for x. Importantly, we solve for the smallest *interval* of x where the condition holds, since x may map to multiple values. This may introduce new reduction variables, as in the following upsampling operation:

```
output(x) = input(x/4);
```

Since x is an integer, 4 values in `input` are used to produce each value of `output`. Accordingly, our converter will generate the following adjoint code:

```
RDom r(0, 4); // loops from 0 to 3
d_output(x) = d_input(4*x + r.x)
```

If any step of this procedure fails to find a solution, we fall back to a general scattering operation. It is still possible to parallelize general scatters using *atomics*. We added atomic operations to Halide's GPU backend to handle this case. A general scatter with atomics usually remains significantly less efficient than our transformed code. For instance, the backward pass of a 2D convolution layer applied to a $16 \times 16 \times 256 \times 256$ input takes 68 ms using atomics and 6 ms with our scatter-to-gather conversion.

Listing 1 shows some derivatives our system would generate for the bilateral slicing example in the left of Figure 4-2.

Differentiating in-place updates

The second case requiring special care arises when an update overwrites some variables of the function, introducing side effects. We categorize the in-place update further into two cases. In the first case the update statements do not reference the variables being overwritten (e.g. `f(x) = 1.f`), and in the second case the overwritten variables are referenced (e.g. `f(x) = 2 * f(x) + 1`).

Differentiating an update without self-reference is simpler. For example, consider the following forward code:

```
g(x) = f(x);
g(1) = 2.f; // update to f that overwrites a value
h(x) = g(x);
```

When backpropagating the adjoints, we need to propagate correctly through the chain of update definitions. While `h(x)` depends on `f(x)` for most `x` (via `g(x)`), this is not true for `x==1`. The update definition to `g` hides the previous dependency on `f(1)`. The corresponding gradient code is:

```
d_g_update(x) = d_h(x); // Propagate to the first update
d_g(x) = d_g_update(x); // Propagate to the initial definition
d_g(1) = 0.f;           // Mask unwanted dependency
d_f(x) = d_g(x);        // Propagate to f
```

In general, if we detect different update arguments between two consecutive function updates (in the example above, `g(1)` is different from `g(x)`), we mask the adjoint of the first update to zero using the update argument of the second update.

In the second case, when the update overwrites an intermediate value, and the intermediate value is required for the derivative, the situation is more complicated. For example:

```
f(x) = g(x)
f(x) = f(x) * f(x)
```

The gradient with respect to `g` requires the overwritten `f(x)`, making it impossible to backpropagate. Following is another example:

```
f(x) = 0
f(x) = 2 * f(x) + g(r.x)
```

In this case the reduction loop `r.x` introduces a dependency between the adjoint of `g(r.x)` and the intermediate `f(x)`. On the other hand, if there is only one self-reference, and the adjoint to that self-reference is 1, then we can differentiate as usual without special treatment:

```
f(x) = 0
f(x) = f(x) + g(r.x)
```

This is because all the intermediate `f(x)` share the same adjoint.

For the first two examples, it is possible to rewrite the forward operation so that the update no longer overwrites intermediate, in a way similar to the stack we used for recording intermediate values in Chapter 2.3.1 and Pearlmutter and Siskind's lambda calculus approach [167]. The first example can be rewritten as:

```
f_(x, 0) = g(x)
f_(x, 1) = f_(x, 0) * f_(x, 0)
f(x) = f_(x, 1)
```

While the second example can be rewritten as:

```
f_(x, 0) = 0
f_(x, r.x + 1) = 2 * f_(x, r.x) + g(r.x)
f(x) = f_(x, r.x.max() + 1)
```

It is possible for the compiler to do the rewrite automatically, but this transformation would change the original algorithm, making manual scheduling more difficult. We opt for more predictive behavior of the compiler. Therefore we detect the following two cases and return an error, asking the user to rewrite the function as above:

- We check if the derivatives depend on a previous value, and if that particular value has been overwritten.
- For updates with reduction variables, unless the derivative of self-reference is 1 or 0, and there is at most one self-reference, we check if the overwritten derivative is used by others.

4.3.3 Checkpointing

Reverse-mode automatic differentiation on complex pipelines must traditionally deal with a difficult trade-off. Memoizing values from the forward evaluation to be reused in the reverse pass saves compute, but costs memory. Even with unlimited memory, bandwidth is limited, so it can be more efficient to recompute values. In automatic differentiation systems, this trade-off is addressed with *checkpointing* [218], which reduces memory usage by recomputing

parts of the forward expressions. Fortunately, this is just a specific instance of the general recomputation-vs-memory trade-off already addressed by Halide's scheduling primitives.

For each function, we can decide whether to create an intermediate buffer for later reuse (the `compute_root` construct), or recompute values at every call site (the `compute_inline` construct). We can also compute these values at some intermediate granularity, i.e., by setting its computation somewhere in the loop nest of their consumers (the `compute_at` construct). Halide also allows checkpointing *across* different Halide pipelines by using a global cache (the `memoize` construct). This is useful when the forward pass and backward pass are in separately-compiled units.

As an example, consider the following 2D convolution implementation in Halide:

```
RDom rk, rt;
convolved(x, y) = 0.f;
convolved(x, y) += in(x - rk.x, y - rk.y) * kernel(rk.x, rk.y);
loss() = 0.f; // define an optimization objective
loss() += pow(convolved(rt.x, rt.y) - target(rt.x, rt.y), 2.f);
auto d = propagate_adjoints(loss);
Func d_in = d(in);
```

We are interested in `d_in`, the gradient of `loss` with respect to `in`. It is given by a cross correlation of `2*(convolved-target)` with `kernel`, where the cross correlation depends on the values of `convolved`. Using the scheduling handles provided by Halide, we can easily decide whether to cache the values of `convolved` for the gradient computation. For example, if we write:

```
convolved.compute_root();
```

the values of `convolved` are computed once and will be fetched from memory when we need them for the derivative `d_in`. On the other hand, if we write:

```
convolved.compute_inline();
```

the values of `convolved` are computed on-the-fly and no buffer is allocated to store them. This can be advantageous when the convolution kernel is small (say 2×1) since this preserves memory locality, or when the pipeline is much longer and we cannot afford to store every intermediate buffer.

Halide provides scheduling primitives that are more general than binary checkpointing decisions. Fine-grained control over the schedule allows exploration of memory/recomputation trade-offs in the forward and gradient code. For instance, we can interleave the computation and storage of `convolved` with the computation of another Halide function that consumes its value (in this case `d_in`). The following code instructs Halide to compute and store a tile of `convolved` for each 32×32 tile of `d_in` computed. This offers a potentially

faster balance between computing all of `convolved` before backpropagation, or recomputing each of its pixels on-demand:

```
d_in.compute_root().tile(x, y, xi, yi, 32, 32);
convolved.compute_at(d_in, x);  // compute at each tile of d_in
```

We timed the three schedules above by computing `d_in`. With multi-threading and vectorization on a CPU, on an image with size of 2560×1600 and kernel size 1×5, the `compute_inline` schedule takes 5.6 milliseconds while the `compute_root` schedule takes 10.1 milliseconds and the `compute_at` schedule takes 9.7 milliseconds. On the same image but with kernel size 3×5, the `compute_inline` schedule takes 66.2 milliseconds while the `compute_root` schedule takes 18.7 milliseconds and the `compute_at` schedule takes 12.3 milliseconds.

4.3.4 Automatic Scheduling

Halide's built-in auto-scheduler [151] navigates performance trade-offs well for stencil pipelines, but struggles with patterns that arise when reversing their computational graph (Chapter 4.3.2). In particular, it does not try to optimize large reductions, like those needed to compute a scalar loss. It also does not generate GPU schedules for the current version of Halide [3]. Therefore we implemented a custom automatic scheduler for gradient pipelines.

Similar to Halide's built-in auto-scheduler, we ask the user to provide an estimate of the input and output buffer sizes. We then infer the extent of all the intermediate functions' domains.

Our automatic scheduler checkpoints (`compute_root`) any stage that scatters or reduces, along with those called by more than one other function. We leave any other functions to be recomputed on-demand (`compute_inline`). For the checkpointed functions, we tile the function domain and parallelize the computation over tiles when possible. Specifically, on CPUs, we split the function's domain into 2D tiles (16×16) and launch CPU threads for each tile, vectorizing the innermost dimension inside a tile. On GPUs, we split the domain into 3D tiles ($16 \times 16 \times 4$). The tiles are mapped to GPU blocks, and elements within a tile to GPU threads. In both cases, we tile the first two (resp. three) dimensions of the function's domain that are large enough. We split the domain if its dimensionality is too low.

If the function's domain is not large enough for tiling, and the function performs a large associative reduction, we transform it into a parallel reduction using Halide's `rfactor` scheduling primitive [203]. This allows us to factorize the reduction into a set of partial

[3] Mullapudi et al.'s work did include experiments on GPU and ARM, but as the Halide compiler has evolved, the original implementation was not able to consistently generate valid schedules.

reductions which we compute in parallel and a final, serial reduction. Like before, we find the first two dimensions of the reduction domain which are large enough for tiling. We reduce the tiles in parallel over CPU threads (resp. GPU blocks). Within each 2D tile, we vectorize (resp. parallelize over GPU threads) the column-wise reductions. We also implemented a multi-level parallel reduction schedule but found it unnecessary in the applications presented. When compiling to GPUs, if both the function domain and the reduction domain are large enough for tiling, but the recursive update does not contain enough pure variables for parallelism, we parallelize the reduction using atomics.

To allow control over checkpointing, the automatic scheduler decisions can be overridden. We ask the user to provide optional lists of Halide functions they do or do not want to inline. We currently do not use `compute_at` in our automatic scheduler.

4.4 Applications and Results

We generate gradients for pipelines in three groups of applications (Figure 4-1). First, we show that our system can be integrated into existing deep learning systems to more easily develop new custom operators. Second, we show that we can improve existing image processing pipelines by optimizing their internal parameters on a dataset of training images. Finally, we show how to use our derivatives to solve inverse imaging problems (i.e., optimizing for the image itself).

Unless otherwise specified, we use our automatic scheduler (Chapter 4.3.4) to schedule all the applications throughout the section (i.e., for both the forward code and the derivatives we generate). Therefore, our implementation only requires the programmer to specify the forward pass of the algorithm.

4.4.1 Custom Neural Network Layers

The class of computations expressible with deep learning libraries such as Caffe [101], PyTorch [165], TensorFlow [1], or CNTK [234] is growing increasingly rich. Nonetheless, it is still common for a practitioner to require a new, custom node tailored to their problem. For instance, TensorFlow offers a bilinear interpolation layer and a separable 2D convolution layer. However, even a simple extension of these operations to 3D would require implementing a new custom operator in C++ or CUDA to be linked with the main library. This can already be tedious and error-prone. Furthermore, while the forward algorithm is being developed, the gradient must be re-derived by hand and kept in sync with the forward operator. This makes experimentation and prototyping especially difficult. Finally, both the forward and

Table 4.1: **Custom neural network operators.** Performance of our approach for custom neural network operators. The runtime measures end-to-end latency for forward+backward evaluation. The spatial transformer transforms a batch of $4 \times 16 \times 512 \times 512$. The Flownet node warps a batch of $4 \times 64 \times 512 \times 512$ images with a 2D warping field. The BilateralSlice layer processes images with size $4 \times 4 \times 1024 \times 1024$ and grid size $4 \times 12 \times 64 \times 64$. Measurements were made on an Intel Core i7-3770K CPU @ 3.50GHz, with 16GB of RAM and an NVIDIA Titan X (Pascal) GPU with 12 GB of RAM.

operator	SpatialTransformer	Flownet	BilateralSlice
PyTorch (cpu)	1094 ms	4240 ms	19819 ms
ours (cpu)	461 ms	2466 ms	1957 ms
PyTorch (gpu)	11 ms	482 ms	1440 ms
CNTK (gpu)	136 ms	404 ms	270 ms
manual CUDA (gpu)	—	181 ms	430 ms
ours (gpu)	13 ms	178 ms	64 ms

backward implementations ought to be reasonably optimized so that a model can be trained in a finite amount of time to verify its design.

We implemented a PyTorch backend for Halide so that our derivatives can be plugged into PyTorch's autograd system. We used this backend to re-implement custom operators recently proposed in the literature: the transformation layer in the spatial transformer network [94], the warping layer in Flownet 2.0 [92], and the bilateral slicing layer in deep bilateral learning [58]. The performance of our automatically scheduled code matches highly-optimized primitives written in CUDA, and is much faster than unoptimized code. We compare the runtime of our method to PyTorch, CNTK, and hand-written CUDA code in Table 4.1.

Spatial transformer network

The spatial transformer network of Jaderberg et al.[94] applies an affine warp to an intermediate feature map of a neural network.

The function containing the forward Halide code is 31 lines long excluding comments, empty lines, and function declarations. Due to the popularity of this operator, deep learning frameworks have implemented specialized functions for the layer. The cuDNN library [39] added its own implementation in version 5 (2016), a year after the original publication. It took another year for PyTorch to implement a wrapper around the cuDNN code. We compare our performance to PyTorch's `grid_sample` and `affine_grid` functions which use the cuDNN implementation on GPU. On 512×512 images with 16 channels and a batch size of 4, our CPU code is around 2.3 times faster than PyTorch's implementation, and our GPU code is around 20% slower than the highly-optimized version implemented in cuDNN. Currently, Halide does not support texture sampling on GPU, which could be causing some of the slowdown. We also compare our performance to a CNTK implementation of spatial

transformer using the *gather* operation. Our GPU code is around 10 times faster than the CNTK implementation.

Having fixed functions such as `affine_grid` can be problematic when users want to slightly modify their models and experiment with different ideas. For example, changing the interpolation scheme (e.g., bicubic or Lanczos instead of bilinear), or interpolating over more dimensions (e.g., transforming volume data) would require implementing a new custom operator. Using our system, these modifications only require minor code changes to the forward algorithm. Our system then generates the derivatives automatically, and our automatic scheduler provides performance without further effort.

Warping layer

FlowNet 2.0 [92], which targets optical flow applications, introduced a new 2D warping layer. Compared to the previous spatial transformer layer, this warping layer is a more general transform using a per-pixel warp-field instead of a parametric transformation.

The function containing the forward Halide code is 18 lines long. The original warping function was implemented as a custom node in Caffe. The authors had to write the forward and reverse code for both the CPU and GPU backends. In total it comprises more than 400 lines of code[4]. While the custom node can handle 2D warps well, adapting it to higher-dimensional warps or semi-parametric warps would be challenging. Our system makes this much easier. In addition to PyTorch and CNTK, we also compare the performance of our GPU code with a highly-optimized reimplementation from NVIDIA[5]. The performance of our code is comparable to the highly-optimized CUDA code.

Bilateral slicing layer

Deep bilateral learning [58] is a general, high-performance image processing architecture inspired by bilateral grid processing and local affine color transforms. It can be used to approximate complicated image processing pipelines with high throughput. The algorithm works by splatting a 2D image onto a 3D grid using a convolutional network. Each voxel of the grid contains an affine transformation matrix. A high-resolution guidance map is then used to *slice* into the grid and produce a unique, interpolated, affine transform to apply to each input pixel. The original implementation in TensorFlow had to implement a custom node[6] for the final slicing operation due to the lack of an efficient way to perform trilinear

[4] FlowNet 2.0: `https://github.com/lmb-freiburg/flownet2/blob/master/src/caffe/layers/flow_warp_layer.cu`

[5] Nvidia FlowNet 2.0: `https://github.com/NVIDIA/flownet2-pytorch`

[6] `https://github.com/mgharbi/hdrnet/blob/master/hdrnet/ops/bilateral_slice.cu.cc`

	kodak	mcm	vdp	moiré	time
bilinear	32.9	32.5	25.2	27.6	*127ms
Adobe Camera Raw 9	33.9	32.2	27.8	29.8	—
AHD [85]	36.1	33.8	28.6	30.8	*1618ms
ours (2 filters, 5x5)	36.7	34.7	29.4	31.5	71ms
ours (9 filters, 5x5)	36.8	35.2	29.8	31.7	177ms
ours (15 filters, 7x7)	37.3	35.5	30.1	32.0	324ms
Gharbi [57]	41.2	39.5	34.3	37.0	2932ms

Table 4.2: **Performance-accurarcy trade-offs.** Peak signal-to-noise ratio for several demosaicking techniques following the evaluation methodology of Gharbi et al. [57] (higher is better). We implemented a version of AHD demosaicking algorithm [85] with our system. Despite the simplicity of our approach, by relaxing the algorithm's specifications (i.e. adding more filters on the green channel reconstruction with larger footprints) and re-optimizing the parameters, we achieve higher fidelity (over 1 dB better) for a similar computational cost. While our method does not rival state-of-the-art deep-learning-based techniques, it is significantly faster and opens up new avenues to optimize more parsimoniously parametrized algorithms tailored to the problem. (Timings reported for a 1 megapixel image. (*)Timing for these algorithms is from non-optimized MATLAB code.)

interpolation on the grid. This custom node also applies the affine transformation on the fly to avoid instantiating a high-resolution image containing all the affine parameters at each pixel. The reference custom node had around 300 lines of CUDA code excluding comments and empty lines. Using the recently introduced general scattering functionality, we can implement the same operation directly in PyTorch. Figure 4-2 shows a comparison between our Halide code, reference CUDA code, and PyTorch code.

PyTorch and CNTK implementations are modestly more complex than our code. PyTorch is 20 times slower while CNTK is 4 times slower on an 1024×1024 input with a grid size of $32 \times 32 \times 8$ and a batch size of 4. CNTK is faster than PyTorch due to different implementation choices on the gather operations. The manual CUDA code aims for clarity more than performance, but is both more complicated and 6.7 times slower than our code.

Gharbi et al. [58] argue that training on high-resolution images is key to capturing the high-frequency features of the image processing algorithm being approximated. Both the PyTorch and CNTK code run out of memory on a 2048×2048 input with grid size $64 \times 64 \times 8$ on a Titan GPU with 12 GB of memory. This makes it almost impossible to experiment with high-resolution inputs. Our code is 13.7 times faster than the authors' reference implementation on this problem size.

4.4.2 Parameter Optimization for Image Processing Pipelines

Traditionally, when developing an image processing algorithm, a programmer manually tunes the parameters of their pipeline to make it work well on a small test set of images.

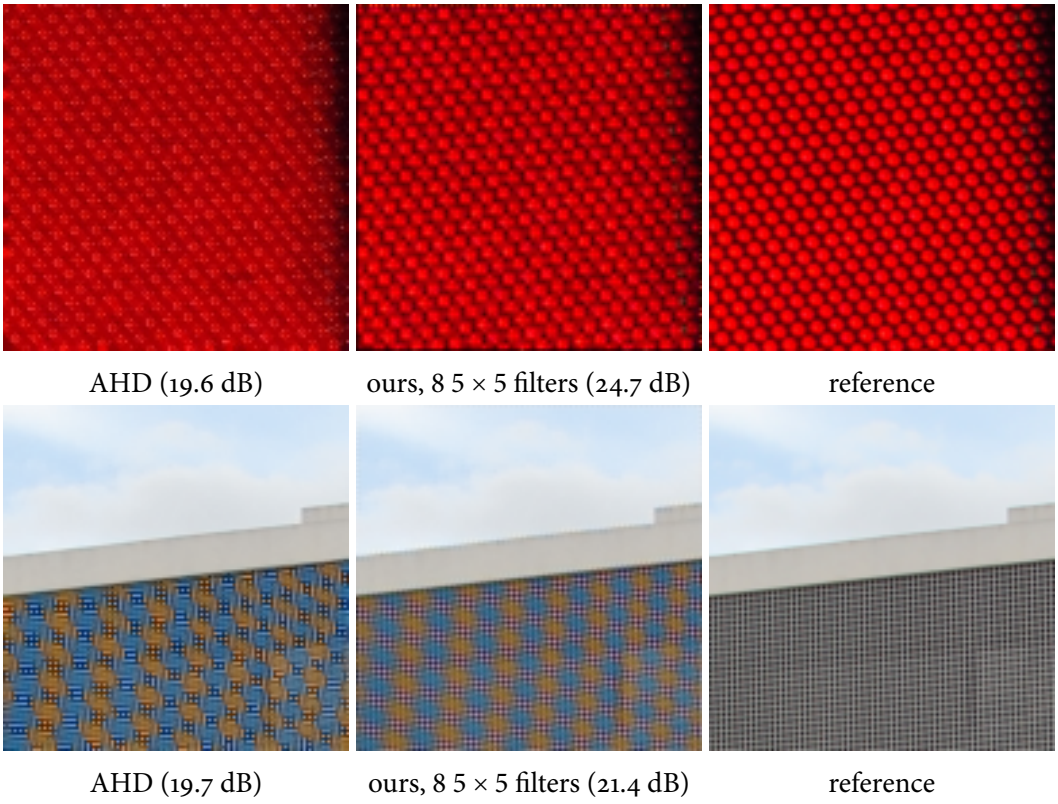
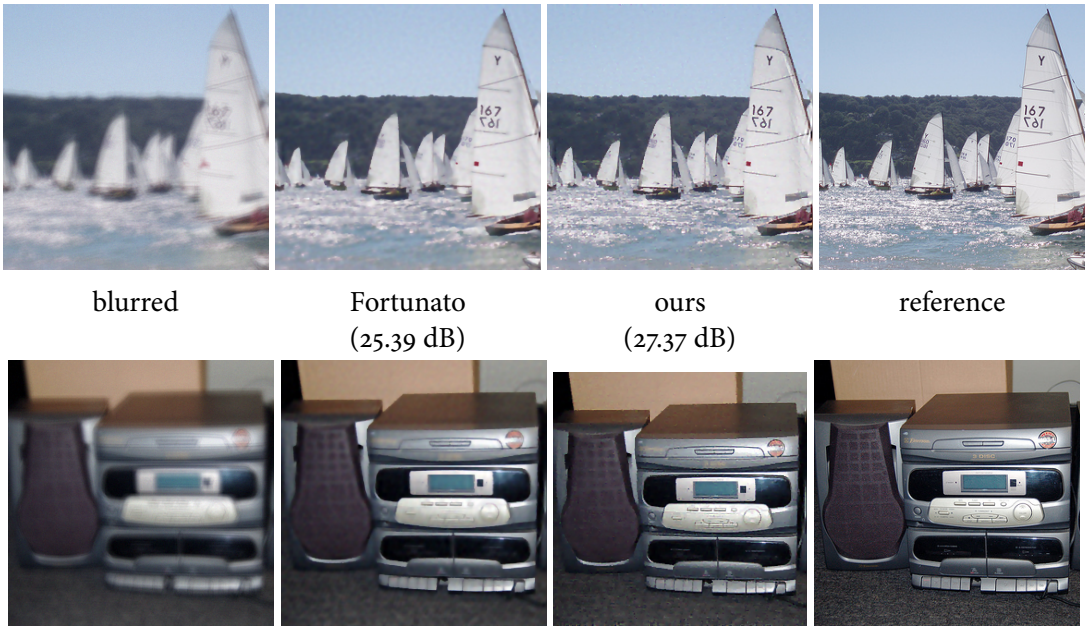


Figure 4-5: **Tuning demosaicking algorithms.** We use our automatic gradients to relax the adaptive homogeneity-directed demosaicing (AHD) algorithm (a) by adding more filters to interpolate the green channel (8 instead of 2 here, with 5x5 footprint instead of 5x1). With this simple tweak, and by optimizing the filters using our automatically generated derivatives, we can obtain sharper images in difficult cases (b), first row. The small-footprint of this simple demosaicking method nevertheless inherits some of the limitations of AHD. In particular, it leads to artifacts in complex, moiré-prone patterns (second row). Images are taken from the deep demosaicking dataset [57].

When the number of parameters is large, manually determining these parameters becomes difficult.

In contrast, modern deep learning methods achieve impressive results by using a large number of parameters and many training images. We demonstrate that it is possible to apply a similar strategy to general image processing algorithms, by augmenting the algorithm with more parameters, and tuning these parameters through an offline training process. Our system provides the necessary gradients for this optimization. Users write the forward code in Halide, and then optimize the parameters of the code using training images.

We demonstrate this with an image demosaicking algorithm based on the adaptive homogeneity directed demosaicking [85] (AHD), and a non-blind image deconvolution algorithm based on sparse adaptive priors [52].



blurred | Fortunato (25.39 dB) | ours (27.37 dB) | reference

blurred | Fortunato (25.83 dB) | ours (27.86 dB) | reference

Figure 4-6: **Tuning deconvolution algorithms.** We use automatic gradients to enhance Fortunato and Oliveira's non-blind deconvolution algorithm [52]. We use more iterations and automatically train the weights, thresholds and filtering parameters. We are able to get sharper results. On eight randomly selected-images we achieve an average PSNR of 29.57 dB. Using the original algorithm with its original parameters the PSNR is 28.51 dB. Image taken from ImageNet [45]

Image demosaicking

Demosaicking seeks to retrieve a full-color image from incomplete color samples captured through a color filter array, where each pixel only contains one out of three red, green and blue colors. Traditional demosaicking algorithms work well on most cases, but can exhibit structured aliasing artifacts such as *zippering* and *moiré* (Figure 4-5). Recent methods using deep learning have achieved impressive results [57], however, the execution time is still an issue for practical usage. We relax the adaptive homogeneity-directed demosaicking algorithm (AHD) [85], variations of which are the default algorithms in *Adobe Camera Raw* and *dcraw*. We increase the number of filters to interpolate the green channel. We also fine-tune the chrominance (red-blue) interpolation filters from the AHD reference. We experiment with different numbers of filters and filter sizes to explore the runtime versus accuracy trade-off. We optimized the filter weights on Gharbi et al.'s [57] training dataset using the gradients provided by our system. The results are illustrated in Table 4.2. With this simple modification, we obtain a significant 1 to 1.5 dB improvement on the more difficult datasets (*moiré* and *vdp*), depending on the number of filters used. We also obtain visually

sharper images in many challenging cases, as shown in Figure 4-5.

With its limited footprint and filtering complexity, our optimized demosaicking still struggles on moiré-prone textures. Our system allows users to experiment with more complex ideas without having to implement the derivatives at each step. For instance, we were able to quickly experiment with (and ultimately discard) alternative algorithms (e.g. using filters that take the ratio between colors into account and 1D directional filters).

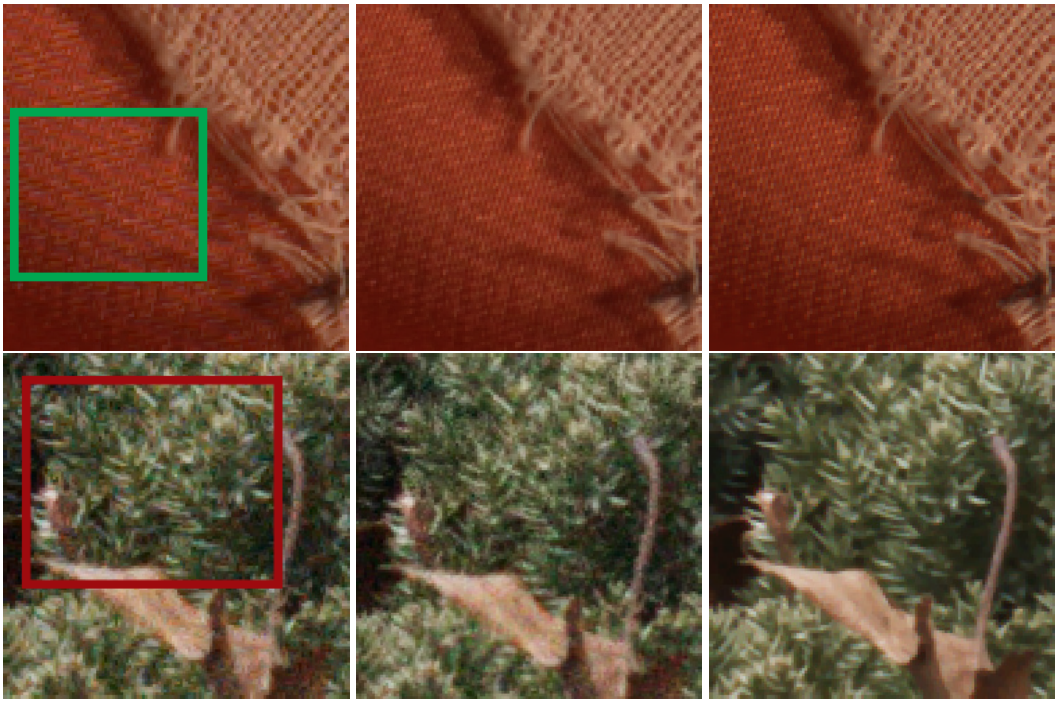
Non-blind image deconvolution

The task of non-blind image deconvolution is: given a point spread function and a blurry image, which is the result of a latent natural image convolved with the function, recover the underlying image. The problem is highly ill-posed, therefore the quality of the reconstruction heavily depends on the priors we place on the image. It is thus important to learn a good set of parameters for those priors.

We based our implementation on the sparse adaptive prior proposed by Fortunato and Oliveira [52]. The original method works in a 2-stage fashion. In the first stage, they solve a conventional L^2 deconvolution using a set of discrete derivative filters as the prior. Then they use an edge-aware filter to clean up the noise in the image. In the second stage, another L^2 deconvolution is solved for large discrete derivatives by matching the prior terms to the result of the first stage, masked by a smooth thresholding function.

We extend the method by increasing the number of stages (we use 4 instead of 2), and having a different set of filters for the priors for each stage. We optimize the weights of the prior filters, the smoothness parameters of the edge-aware filter (we use a bilateral grid), and the thresholding parameters in the smooth thresholding functions.

To demonstrate the ability of our system to handle nested derivatives, we implemented a generic conjugate gradient solver using a linear search algorithm based on Newton-Raphson to solve for the L^2 deconvolution. We write the conjugate gradient loop in PyTorch, but implement the gradient and vector-Hessian-vector product (required in the line search step) in Halide. We also implemented the bilateral grid filtering step in Halide. To optimize the parameters, we then differentiate through the gradients we used for the non-linear conjugate gradient algorithm. We train our method on ImageNet [45] and use the point spread function generation scheme described in Kupyn et al.'s work [119]. We initialize the parameters to the recommended parameters described in Fortunato and Oliveira's work. Figure 4-6 shows the result.



(a) dcraw (AHD) single frame

(b) Gharbi et al. [57] single frame

(c) our output R

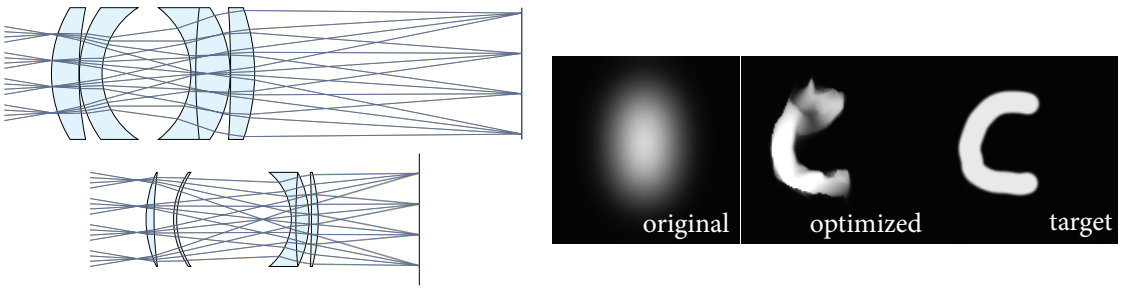
Figure 4-7: **Gradients for inverse problems.** Automatic gradients can be used for inverse problems such as high-resolution demosaicking from a burst of images. The user only needs to implement the forward model. Bursts of RAW images are captured with a *Nikon D810* camera then jointly aligned and demosaicked (13 and 23 images respectively, only showing crops). We initialize our reconstruction to a simple bilinear interpolation (not shown) and solve an inverse problem to recover both a set of homographies and a demosaicked image that matches the captured data when reprojected. Compared to the result of *dcraw*'s AHD algorithm (a) and Gharbi et al. [57] (b), our output (c) is much sharper, and shows less noise (red square) and color moiré (green square).

4.4.3 Inverse Imaging Problems: Optimizing for the Image

The derivatives produced by our automatic differentiation algorithm can be readily employed to solve inverse problems in computational photography. Using our system, users can quickly experiment with different forward models or different priors. We demonstrate this on a burst-demosaicking inverse pipeline.

Given N misaligned Bayer RAW images, our goal is to reconstruct a full-color image as well as estimate the homography parameters that align our reconstruction to the input data. We do this by minimizing the following cost function:

$$\min_{R,H_i} \sum_{i=1}^{N} ||MH_iR - I_i||_2^2 + \lambda||\nabla R||_1 \tag{4.1}$$



(a) lens optimization (b) inverse fluid simulation

Figure 4-8: **Non-image-processing applications.** Halide augmented with gradients is useful for a wider range of applications than just image processing and machine learning. (a) By expressing a ray tracer for an optical system in Halide and taking derivatives of sharpness with respect to the lens parameters, we can reoptimize a classic Zeiss lens design [125] (above) to be more compact (below) while maintaining as much sharpness as possible. (b) We can also optimize for fluid simulation, by taking a key frame from an original animation, and optimizing it to be as similar as possible to the target. We implement stable fluid [201] in Halide and optimize for the force field per frame, diffusion constant, viscosity, and time step size to make the last frame of the animation match the target.

where M decimates the color samples according to the Bayer mosaic pattern. The homographies H_i align our reconstruction R to the input data I_i.

Gradient descent can help us minimize the function locally, but Equation 4.1 is highly non-convex, so a good initialization is critical. We initialize the H_i using RANSAC [51] and SIFT-based features [139] in a pairwise fashion. We also initialize $R = I_0$. This part is implemented in OpenCV[7]. From this starting point, we jointly refine the alignment and our estimate of the full-color image by minimizing the loss function (4.1). Compared to any individual image I_i, our reconstruction is sharper, and does not suffer from color *moiré* artifacts (Figure 4-7). We use the Adam gradient-descent optimizer [115] for 300 iterations, setting the learning rate to 10^{-2} for R and 10^{-4} for H_i. Our algorithm provides the gradient of the loss with respect to the reconstructed image R and homographies H_i. We set $\lambda = 10^{-3}$. For 13 2048 × 2048 images, computing the initial homographies takes 44.5s, initializing the reconstruction 0.1s. Minimizing the cost function takes 179.4s using the code generated by our automatic scheduler on a Titan X (Pascal) GPU.

4.4.4 Non-image-processing Applications

While we focus on image processing, Halide can express any feed-forward pipeline of arithmetic on multi-dimensional arrays (Figure 4-8). There are numerous non-imaging applications in this class, and taking derivatives is useful for many of them. We implement two

[7] OpenCV: https://github.com/opencv/opencv

examples of this. First, we implemented a simple ray-tracer for a system of spherical lenses in Halide, and used our system to construct derivatives of the sharpness with respect to the lens positions and curvatures. In Figure 4-8a, we start from an existing Zeiss design [125] and re-optimize it to be more compact while maintaining the field of view, F-number, and sharpness.

Secondly, we implement a classical grid-based fluid simulation algorithm [201] and use our system to differentiate the whole fluid simulation process. We implement a fluid control system (Figure 4-8b). Given a target keyframe and an initial sequence of simulation images, we try to find a source force that "bends" the fluid to the desired image. We optimize for the force fields per frame, diffusion constant, viscosity, and time step size. This is not a novel application, but previously the derivatives were hand-derived [210, 148], while our system is capable of generating derivatives automatically, and adapt to new simulation algorithms.

4.4.5 Future Work

As these applications demonstrate, our system automatically delivers state of the art performance when computing the gradients of general image processing pipelines. We see three major directions for future work.

Higher-order derivatives and non-scalar outputs. Some optimization methods require derivatives of non-scalar outputs, the full Hessian matrix, or even higher-order derivatives [60]. Our system supports repeated or nested application of differentiation. However, it only differentiates with respect to one scalar at a time. When the dimensionality of both the input and the output are high, there are automatic differentiation algorithms that are more efficient than both forward- and reverse-mode (Chapter 2.2.3). Incorporating these algorithms into our system, and developing better interfaces for non-scalar outputs and higher-order derivatives, will broaden the range of possible applications.

Better automatic scheduling. While it is possible to manually schedule the synthesized reverse computation, we found it challenging for non-trivial examples, and relied on our automatic scheduler entirely for this work. Its performance is good for gradient pipelines, but inspecting the generated code reveals plenty of room for further improvement. We consider the general Halide automatic scheduling problem still unsolved.

More general programming model. Halide assumes all operations are performed on a multi-dimensional grid. While this is a rather general programming model, there are many operations outside of image processing that are ill-suited for this model, such as sparse

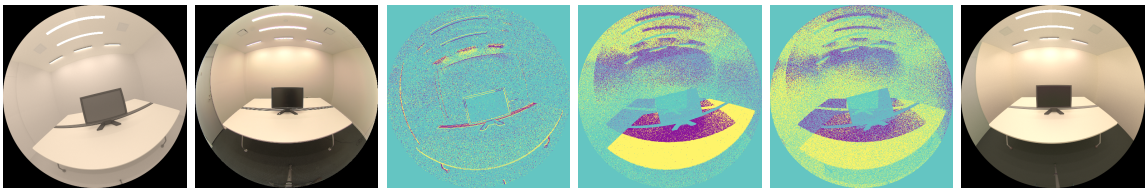
matrix multiplication or tree traversal. Generalizing Halide to handle differentiation of these operations, or developing new differentiable programming language to explore different trade-offs are both interesting directions.

4.5 Conclusion

Gradient-based optimization is revolutionizing many fields including image processing, but efficient computation of derivatives has so far been difficult, requiring one to either conform to limited building blocks or to error-prone manual derivation and challenging performance tuning. In contrast, our method can automatically generate high-performance gradient code for general image processing pipelines. Our method only requires the implementation of the operators in a language that is concise, easy to maintain, and portable. It then automatically derives the gradient code using reverse automatic differentiation. We have presented a new automatic performance tuner that handles the particular computation patterns exhibited by derivatives. Our code compiles to a variety of platforms such as x86, ARM, and GPUs, which is critical both for final deployment and for efficient training.

We have demonstrated that our work enables several types of applications, from custom neural network nodes, to the tuning of internal image processing parameters, to the solution of inverse problems. It dramatically simplifies the exploration of custom neural network nodes by automatically providing a level of performance that has so far been reserved to advanced CUDA programmers. It makes it easy to optimize internal weights and parameters for general image-processing pipelines, a step that few practitioners feel they can afford due to the cost of implementing gradients, which is especially true during the algorithmic exploration stages. Our system can also be used for inverse problems (which can even include unknown imaging parameters in addition to the unknown image). The user now only needs to worry about implementing the forward model. Each of the demonstrated applications was implemented initially in a few hours, and then evolved rapidly, with correct gradients and high-performance implementation automatically provided at each step by our method. We believe this will create new opportunities for rapid research and development in learning- and optimization-based imaging applications.

5 | Differentiable Monte Carlo Ray Tracing through Edge Sampling



(a) initial guess (b) photograph (c) camera gradient (d) table albedo gradient (e) light gradient (f) our fitted result

Figure 5-1: **Differentiable Monte Carlo ray tracing.** We develop a general-purpose differentiable renderer that is capable of handling general light transport phenomena. Our method generates gradients with respect to scene parameters, such as camera pose (c), material parameters (d), mesh vertex positions, and lighting parameters (e), from a scalar loss computed from the output image. (c) shows the per-pixel gradient contribution of the L^1 difference with respect to the camera moving into the screen. (d) shows the gradient with respect to the red channel of table albedo. (e) shows the gradient with respect to the green channel of the intensity of one light source. As one of our applications, we use our gradient to perform an inverse rendering task by matching a real photograph (b) starting from an initial configuration (a) with a manual geometric recreation of the scene. The scene contains a fisheye camera with strong indirect illumination and non-Lambertian materials. We optimize for camera pose, material parameters, and light source intensity. Despite slight inaccuracies due to geometry mismatch and lens distortion, our method generates an image (f) that almost matches the photo reference.

The increasing importance of derivatives-based optimization creates a need for rendering algorithms that can be differentiated with respect to arbitrary input parameters, such as camera location and direction, scene geometry, lights, material appearance, or texture values. Unfortunately, the rendering integral includes visibility terms that are not differentiable at object boundaries. Whereas the final image function is usually differentiable once radiance has been integrated over pixel prefilters, light source areas, etc., the integrand of rendering algorithms is not. In particular, the derivative of the integrand has Dirac delta terms at occlusion boundaries that cannot be handled by traditional sampling strategies.

Previous work in differentiable rendering [137, 109] has focused on fast, approximate

solutions using simpler rendering models that only handle primary visibility, and ignore secondary effects such as shadows and indirect light. Analytical solutions exist for diffuse interreflection [9] but are difficult to generalize for arbitrary material models. The work by Ramamoorthi et al. [173] is an exception but it only differentiates with respect to image coordinates, whereas we want derivatives with respect to arbitrary scene parameters. Other previous work usually also relies on finite differences, with the usual limitation of these methods when the function is complex, namely that these methods work well for simple configurations but they do not propose a comprehensive solution to the full light transport equation.

In this chapter, we propose an algorithm that is, to the best of our knowledge, the first to compute derivatives of scalar functions over a physically-based rendered image with respect to arbitrary input parameters (camera, light materials, geometry, etc.). Our solution is stochastic and builds on Monte Carlo ray tracing [105]. For this, we introduce new techniques to explicitly sample edges of triangles in addition to the usual solid angle sampling of traditional approaches. This requires new spatial acceleration strategies and importance sampling to efficiently sample edges. Our method is general and can sample derivatives for arbitrary bounces of light transport. The running times we observed range from a second to a minute depending on the required precision, for an overhead of roughly $10\times - 20\times$ compared to rendering an image alone.

We integrate our differentiable ray tracer with the automatic differentiation library PyTorch [165] for efficient integration with optimization and learning approaches. The scene geometry, lighting, camera, and materials are parameterized by PyTorch tensors, which enables a complex combination of 3D graphics, light transport, and neural networks. Backpropagation runs seamlessly across PyTorch and our renderer.

5.1 Related Work

5.1.1 Inverse Graphics

Inverse graphics techniques seek to find the scene parameters given observed images. Vision as inverse graphics has a long history in both computer graphics and vision (e.g. [14, 235, 166]). Many techniques in inverse graphics use derivatives of the rendering process for inference.

Blanz and Vetter [24] optimized for the shape and texture of a face. Shacked and Lischinski [194] and Bousseau et al. [25] optimized a perceptual metric for lighting design. Gkioulekas et al. [62, 61] focused on scattering parameters. Aittala et al. [4, 5, 3] inferred spatially varying material properties. Barron et al. [12] proposed a solution to jointly optimize shape,

illumination, and reflectance. Khungurn et al. [114] and Zhao et al. [242] concentrated on optical properties of fabrics. All of the above approaches use gradients for solving the inverse problem, and had to develop a specialized solver to compute the gradient of the specific light transport scenarios they were interested in.

Loper and Black [137] and Kato et al. [109] proposed two general differentiable rendering pipelines. Rhodin et al. [177] developed a differentiable volumetric ray caster. Liu et al. [134] differentiated rendering process for inverse geometry editing. Athalye et al. [10], Zeng et al. [237], and Liu et al. [135] all use differentiable rasterizers for adversarial example synthesis. All of them focus on performance and approximate the primary visibility gradients when there are multiple triangles inside a pixel, and assume Lambertian materials and do not compute shadows and global illumination.

Recently, it is increasingly popular for deep learning methods to incorporate a *differentiable rendering layer* in their architecture (e.g. [133, 178]). These rendering layers are usually special purpose and do not handle geometric discontinuities such as primary visibility and shadow.

To our knowledge, our method is the first that is able to differentiate through a full path tracer, while taking the geometric discontinuities into account.

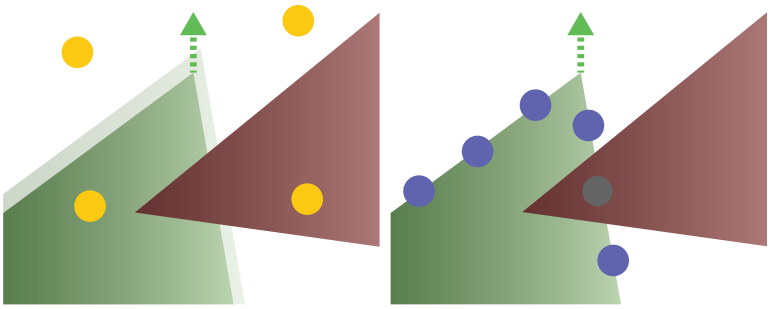
5.1.2 Derivatives in Rendering

Analytical derivatives have been used for computing the footprint of light paths [196, 90, 204] and predicting the changes of specular light paths [37, 97, 106]. The derivatives are usually manually derived for the particular type of light paths the work focused on, making it difficult to generalize to arbitrary material models or lighting effects. Unlike these methods, we compute the gradients using a hybrid approach that mixes automatic differentiation and manually derived derivatives focusing on the discontinuous integrand.

Arvo [9] proposed an analytical method for computing the spatial gradients for irradiance. The method requires clipping of triangle meshes in order to correctly integrate the form factor, and does not scale well to scenes with large complexity. It is also difficult or impossible to compute closed-form integration for arbitrary materials.

Ramamoorthi et al.'s work on first order analysis of light transport [173] is highly related to our method. Their method is a special case of ours. Our derivation generalizes their method to differentiate with respect to any scene parameters. Furthermore, we handle primary visibility, secondary visibility, and global illumination.

Irradiance or radiance caching [224, 117, 99] numerically computes the gradient of inter-reflection with respect to spatial position and orientation of the receiver. To take disconti-



(a) area sampling (b) edge sampling

Figure 5-2: **Area sampling v.s. edge sampling.** (a) The figure shows a pixel overlapped with two triangles. We are interested in computing the derivative of pixel color with respect to the green triangle moving up. Since the area covered by the green triangle increases, the final pixel color will contain more green area and less white background. Traditional area sampling (yellow samples) even instrumented with automatic differentiation does not account for the change in the covered area. (b) In addition to traditional area sampling, we propose a novel edge sampling algorithm (blue samples) to sample the differential area on the edges. Our method computes unbiased gradients and correctly takes occlusion into account.

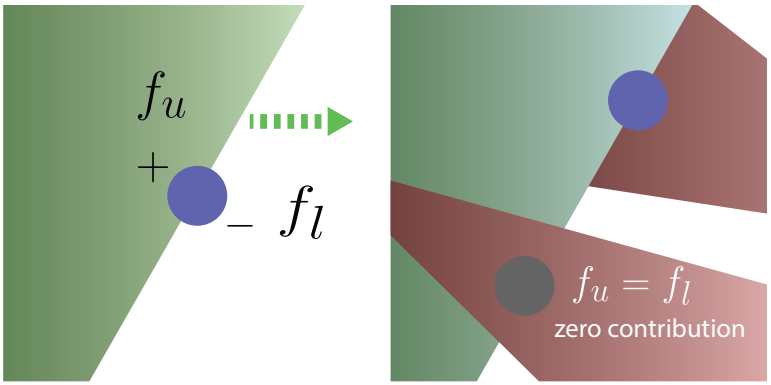
nuities into account, these methods resort to stratified sampling. Unlike these methods, we estimate the gradient integral directly by automatic differentiation and edge sampling.

In Chapter 6, we propose a variant of the Metropolis light transport [216] algorithm by computing the Hessian of a light path contribution with respect to the path parameters using automatic differentiation. It focuses on second-derivatives for forward rendering and does not take geometric discontinuities into account. We will discuss more in Chapter 6.

5.2 Method

Our task is the following: given a 3D scene with a continuous parameter set Φ (including camera pose, scene geometry, material and lighting parameters), we generate an image using the path tracing algorithm [105]. Given a scalar function computed from the image (e.g. a loss function we want to optimize), our goal is to backpropagate the gradient of the scalar with respect to all scene parameters Φ.

The pixel color is formalized as an integration over all light paths that pass through the pixel filter. We use Monte Carlo sampling to estimate both the integral and the gradient of the integral [213, 168]. However, since the integrand is discontinuous due to edges of geometry and occlusion, traditional area sampling does not correctly capture the changes due to camera parameters or triangle vertex movement (Figure 5-2 a). Mathematically, the gradient of the discontinuous integrand is a Dirac delta function, therefore traditional sampling has zero



(a) half-spaces (b) occlusion

Figure 5-3: **Edge sampling.** (a) An edge splits the space into two half-spaces f_u and f_l. If the edge moves right, the green area increases while the white area decreases. We integrate over edges to compute gradients by taking into account the change in areas. To compute the integration, we sample a point (the blue point) on the edge and compute the difference between the half-spaces by computing the color on the two sides of the edge. (b) Our method handles occlusion correctly since an occluded sample will land on the continuous part of the path contribution function, thus having the exact same contribution on the two sides (for example, the grey sample has zero contribution to the gradient).

probability of capturing the Dirac deltas.

Our strategy for computing the gradient integral is to split it into smooth and discontinuous regions (Figure 5-2). For the smooth part of the integrand (e.g. Phong shading or trilinear texture reconstruction), we employ traditional area sampling with automatic differentiation. For the discontinuous part, we use a novel edge sampling method to capture the changes at boundaries. In this section, we first focus on primary visibility where we only integrate over the 2D screen-space domain (Chapter 5.2.1). We then generalize the method to handle shadow and global illumination (Chapter 5.2.2).

We focus on triangle meshes and we assume the meshes have been preprocessed such that there is no interpenetration. We also assume no point light sources and no perfectly specular surfaces. We approximate these with area light sources and materials with very low roughness. We also focus on static scenes and leave integration over the time dimension for motion blur as future work.

5.2.1 Primary Visibility

We start by focusing on the 2D pixel filter integral for each pixel that integrates over the pixel filter k and the radiance L, where the radiance itself can be another integral that integrates over light sources or the hemisphere. We also focus on linear projective cameras for simplicity. We will generalize the method to handle discontinuities inside the radiance integral in Chapter 5.2.2. We will also generalize to nonlinear projections such as fisheye cameras in

Chapter 5.2.3. The pixel color I can be written as:

$$I = \iint k(x, y)L(x, y)dxdy. \tag{5.1}$$

For notational convenience, we will combine the pixel filter and radiance and call them scene function $f(x, y) = k(x, y)L(x, y)$. We are interested in the gradients of the integral with respect to some parameters Φ in the scene function $f(x, y; \Phi)$, such as the position of a mesh vertex or camera pose:

$$\nabla I = \nabla \iint f(x, y; \Phi)dxdy. \tag{5.2}$$

The integral usually does not have a closed-form solution, especially when more complex effects such as non-Lambertian materials are involved. Therefore we rely on Monte Carlo integration to estimate the pixel value I:

$$I \approx \frac{1}{N} \sum_i f(x_i, y_i; \Phi), \tag{5.3}$$

where N is the number of samples we use for pixel I, and x_i, y_i are the screen-space samples. Unfortunately, we cannot take the naive approach of applying the same Monte Carlo estimator for the gradient ∇I, since the scene function f is not necessarily differentiable with respect to the scene parameters (Figure 5-2a).

A key observation is that all the discontinuities happen at triangle edges, since we assume no interpenetration. This allows us to explicitly integrate over the discontinuities. A 2D triangle edge splits the space into two half-spaces (f_u and f_l in Figure 5-3). We can model it as a Heaviside step function θ:

$$\theta(\alpha(x, y))f_u(x, y) + \theta(-\alpha(x, y))f_l(x, y), \tag{5.4}$$

where f_u represents the upper half-space, f_l represents the lower half-space, and α defines the edge equation formed by the triangles.

For each edge with two endpoints $(a_x, a_y), (b_x, b_y)$, we can construct the edge equation by forming the line $\alpha(x, y) = Ax + By + C$, since we assume we are using a projective camera. If $\alpha(x, y) > 0$ then the point is in the upper half-space, and vice versa. For the two endpoints of the edge $\alpha(x, y) = 0$. Hence, by plugging in the two endpoints we obtain:

$$\alpha(x, y) = (a_y - b_y)x + (b_x - a_x)y + (a_x b_y - b_x a_y). \tag{5.5}$$

We can rewrite the scene function f as a summation of Heaviside step functions θ with edge equation α_i multiplied by an arbitrary function f_i:

$$\iint f(x,y)dxdy = \sum_i \iint \theta(\alpha_i(x,y))f_i(x,y)dxdy. \tag{5.6}$$

f_i itself can contain Heaviside step functions: for example, a triangle defines a multiplication of three Heaviside step functions. Occlusion can also be modeled by multiplying step functions from other edges closer to the camera. f_i can even be an integral over light sources or the hemisphere. This is crucial for our later generalization to secondary visibility.

We want to analytically differentiate the Heaviside step function θ and explicitly integrate over its derivative – the Dirac delta function δ. To do this we first swap the gradient operator inside the integral, then we use the product rule to separate the integral into two:

$$\begin{aligned} &\nabla \iint \theta(\alpha_i(x,y))f_i(x,y)dxdy \\ &= \iint \delta(\alpha_i(x,y))\nabla\alpha_i(x,y)f_i(x,y)dxdy + \iint \nabla f_i(x,y)\theta(\alpha_i(x,y))dxdy. \end{aligned} \tag{5.7}$$

Equation (5.7) shows that we can estimate the gradient using two Monte Carlo estimators. The first one estimates the integral over the edges of triangles containing the Dirac delta functions, and the second estimates the original pixel integral except the smooth function f_i is replaced by its gradient, which can be computed through automatic differentiation.

To estimate the integral containing Dirac delta functions, we eliminate the Dirac function by performing a variable substitution to rewrite the first term containing the Dirac delta function to integrate over the edge, that is, over the regions where $\alpha_i(x,y) = 0$:

$$\iint \delta(\alpha_i(x,y))\nabla\alpha_i(x,y)f_i(x,y)dxdy = \int_{\alpha_i(x,y)=0} \frac{\nabla\alpha_i(x,y)}{\left\|\nabla_{x,y}\alpha_i(x,y)\right\|} f_i(x,y)d\sigma(x,y), \tag{5.8}$$

where $\left\|\nabla_{x,y}\alpha_i(x,y)\right\|$ is the L^2 length of the gradient of the edge equations α_i with respect to x, y, which takes the Jacobian of the variable substitution into account. $\sigma(x,y)$ is the measure of the length on the edge [88].

The gradients of the edge equations α_i are:

$$\begin{aligned}
&\left\|\nabla_{x,y}\alpha_i\right\| = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \\
&\frac{\partial \alpha_i}{\partial a_x} = b_y - y,\ \frac{\partial \alpha_i}{\partial a_y} = x - b_x \\
&\frac{\partial \alpha_i}{\partial b_x} = y - a_y,\ \frac{\partial \alpha_i}{\partial b_y} = a_x - x \\
&\frac{\partial \alpha_i}{\partial x} = a_y - b_y,\ \frac{\partial \alpha_i}{\partial y} = b_x - a_x.
\end{aligned} \tag{5.9}$$

As a byproduct of the derivation, we also obtain the screen space gradients $\frac{\partial}{\partial x}$ and $\frac{\partial}{\partial y}$, which can potentially facilitate adaptive sampling as shown in Ramamoorthi et al.'s first-order analysis [173]. We can obtain the gradient with respect to other parameters, such as camera parameters, 3D vertex positions, or vertex normals by propagating the derivatives from the projected triangle vertices using the chain rule:

$$\frac{\partial \alpha}{\partial p} = \sum_{k \in \{x,y\}} \frac{\partial \alpha}{\partial a_k}\frac{\partial a_k}{\partial p} + \frac{\partial \alpha}{\partial b_k}\frac{\partial b_k}{\partial p}, \tag{5.10}$$

where p is the desired parameter.

We use Monte Carlo sampling to estimate the Dirac integral (Equation (5.8)). Recall that a triangle edge defines two half-spaces (Equation (5.4)), therefore we need to compute the two values $f_l(x, y)$ and $f_u(x, y)$ on the edge (Figure 5-3). By combining Equation (5.4) and Equation (5.8), our Monte Carlo estimation of the Dirac integral for a single edge E on a triangle can be written as:

$$\frac{1}{N}\sum_{j=1}^{N} \frac{\|E\|\,\nabla\alpha_i(x_j, y_j)(f_u(x_j, y_j) - f_l(x_j, y_j))}{P(E)\left\|\nabla_{x_j,y_j}\alpha_i(x_j, y_j)\right\|}, \tag{5.11}$$

where $\|E\|$ is the length of the edge and $P(E)$ is the probability of selecting edge E.

In practice, we use a path tracer to evaluate the values on the two sides of an edge ($f_l(x, y)$ and $f_u(x, y)$). We trace two light paths from the screen space position (x, y), and offset them by a small amount (say, 10^{-6}). We use the same random number sequence for both light paths to minimize the variance through correlated sampling. Note that this is different from finite differences: we do not move the target parameter to acquire their derivatives.

If we employ smooth shading, most of the triangle edges are in the continuous regions and the Dirac integral is zero for these edges since by definition of continuity $f_u(x, y) = f_l(x, y)$. Only the *silhouette edges* (e.g. [83]) have non-zero contribution to the gradients. We select

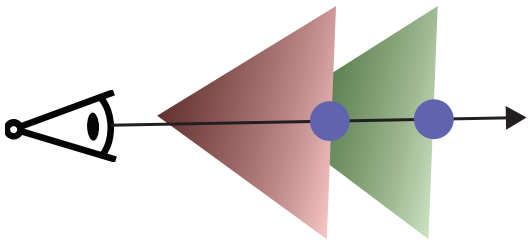
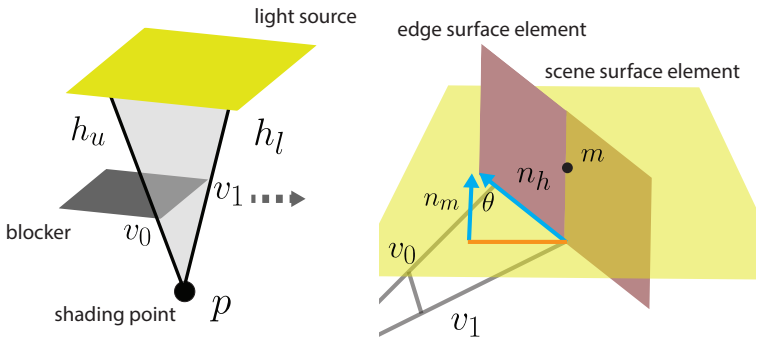


Figure 5-4: **Parallel edges.** Our method almost always handles occlusion correctly. However, it can produce incorrect results in the pathological case where two edges are exactly parallel to the viewport. If we sampled the occluded parallel edge, we consider it occluded and wrongly ignore the color of the occluded triangle. If we sampled the edge that occludes other parallel edges, we do not take the occluded triangle color into consideration. Fortunately, this is a zero-measure case and it rarely happens in practice.

the edges by projecting all triangle meshes to screen space and clip them against the camera frustum. We select one silhouette edge with probability proportional to the screen space lengths. We then uniformly pick a point on the selected edge. For thin-lens camera that produce depth-of-field effects, we use the center of the camera pupil as the basis of projection, and we conservatively test the silhouette using the four corners of the pupil bounding box.

Our method handles occlusion correctly, since if the sample is blocked by another surface, (x, y) will always land on the continuous part of the contribution function $f(x, y)$. Such samples will have zero contribution to the gradients. Figure 5-3b illustrates the process. However, in the pathological case where two edges are exactly parallel to the viewport (Figure 5-4), we can compute the wrong result. This is because occlusion is modeled as a multiplication between two Heaviside step functions in Equation (5.6), and if two step functions coincide exactly, we completely mask the occluded edge and wrongly ignore its contribution. In theory, this can be resolved by detecting parallel edges in the ray casting operation, and properly breaking even: If we sampled an edge occluded by a parallel edge, treat it as not occluded. On the other hand, if we sampled an edge that occludes other parallel edges, set the other half-space to the occluded triangle. However, since this is a zero-measure case and we never observe it in practice, we do not implement it at the moment.

To recap, we use two sampling strategies to estimate the gradient integral of pixel filter (Equation (5.2)): one for the discontinuous regions of the integrand (first term of Equation (5.7)), one for the continuous regions (second term of Equation (5.7)). To compute the gradient for discontinuous regions, we need to explicitly sample the edges. We compute the difference between two sides of the edges using Monte Carlo sampling (Equation (5.11)).



(a) secondary visibility (b) width correction

Figure 5-5: (a) Our method can be easily generalized to handle shadow and global illumination. Similar to the primary visibility case (Figure 5-3), a geometry edge (v_0, v_1) and the shading point p splits the 3D space into two half-spaces f_u and f_l and introduces a discontinuity. Assuming the blocker is moving right, we integrate over the edge to compute the difference. By doing so we take account of the increase in blocker area and the decrease in light source area looking from the shading point. The integration over the edge is defined on the intersection between the scene manifold and the plane formed by the shading point and the edge (the semi-transparent triangle). (b) The orientation of the infinitesimal width of the edge differs from the scene surface element the edge intersects with. During the integration, we need to project the scene surface element width onto the edge surface element. The ratio of the widths between the two is determined by $\frac{1}{\sin\theta}$, which is one over the length of the cross product between the normal of the edge plane and the scene surface.

5.2.2 Secondary visibility

Our method can be easily generalized to handle effects such as shadow and global illumination by integrating over the 3D scene. Figure 5-5 illustrates the idea.

We focus on a single shading point p since we can propagate the derivatives back to screen space and camera parameters using Equation (5.7). Given the shading point, the shading equation involves integration over all points m on the scene manifold $\mathcal{M}$:

$$g(p) = \int_{\mathcal{M}} h(p, m) dA(m), \tag{5.12}$$

where A is the area measure of point m, and h is the scene function including material response, geometric factor, incoming radiance, and visibility. Note that $g(p)$ can itself be part of the pixel integrand $f(x, y)$ in the previous section (Equation (5.1)). Therefore we can propagate the gradient of $g(p)$ using the chain rule or automatic differentiation with Equation (5.7).

Similar to the primary visibility case, an edge (v_0, v_1) in 3D introduces a step function

into the scene function h:

$$\theta(\alpha(p,m))h_u(p,m) + \theta(-\alpha(p,m))h_l(p,m). \tag{5.13}$$

We can derive the edge function $\alpha(m)$ by forming a plane using the shading point p and the two points on the edge. The sign of the dot product between the vector $m - p$ and the plane normal determines the two half-spaces. Therefore, the edge equation $\alpha(m)$ can be defined by

$$\alpha(p,m) = (m-p)\cdot(v_0-p)\times(v_1-p)\,. \tag{5.14}$$

To compute the gradients, we analogously apply the derivation used for primary visibility, using the 3D version of Equation (5.7) and Equation (5.8) with x, y replaced by p, m. The edge integral integrating over the line on the scene surface, analogous to Equation (5.8) is:

$$\begin{aligned}&\int_{\alpha(p,m)=0} \frac{\nabla\alpha(p,m)}{\|\nabla_m\alpha(p,m)\|} h(p,m) \frac{1}{\|n_m \times n_h\|} d\sigma'(m)\\ &n_h = \frac{(v_0-p)\times(v_1-p)}{\|(v_0-p)\times(v_1-p)\|},\end{aligned} \tag{5.15}$$

where n_m is the surface normal on point m. There are two crucial differences between the 3D edge integral (Equation (5.15)) and the previous screen space edge integral (Equation (5.8)). First, while the measure of the screen space edge integral $\sigma(x,y)$ coincides with the unit length of the 2D edge, the measure of the 3D edge integral $\sigma'(m)$ is the length of projection of a point on the edge from the shading point p to a point m on the scene manifold (the semi-transparent triangle in Figure 5-5a illustrates the projection). Second, there is an extra area correction term $\|n_m \times n_h\|$, since we need to project the scene surface element onto the infinitesimal width of the edge (Figure 5-5b).

To integrate the 3D edge integral using Monte Carlo sampling we substitute the variable again from the point m on the surface to the line parameter t on the edge $v_0 + t(v_1 - v_0)$:

$$\int_0^1 \frac{\nabla\alpha(p,m(t))}{\|\nabla_m\alpha(p,m(t))\|} h(p,m(t)) \frac{\|J_m(t)\|}{\|n_m \times n_h\|} dt, \tag{5.16}$$

where the Jacobian $J_m(t)$ is a 3D vector describing the projection of edge (v_0, v_1) onto the scene manifold with respect to the line parameter. We derive the Jacobian in Appendix 5.A.

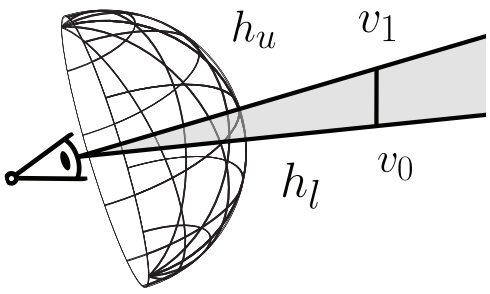


Figure 5-6: **Cameras with non-linear projection.** Our method can also be extended to work with cameras with non-linear projection, such as a fisheye camera as shown in this figure. Gradients of these camera models cannot be computed directly using the 2D edge sampling described in Section 5.2.1 and Figure 5-3, because the projection of edges are not straight lines. Instead of sampling the projected edge in screen space, we directly sample the edge (v_0, v_1) in the 3D space, and use the 3D step function as in the secondary visibility case to split the spaces into two half-spaces h_u and h_l (Section 5.2.2 and Figure 5-5).

The derivatives for $\alpha(p, m)$ needed to compute the edge integral are:

$$\begin{aligned}
&{v_0}' = v_0 - p, {v_1}' = v_1 - p, m' = m - p \\
&\|\nabla_m \alpha(p, m)\| = \|{v_0}' \times {v_1}'\| \\
&\nabla_{v_0} \alpha(p, m) = {v_1}' \times m' \\
&\nabla_{v_1} \alpha(p, m) = m' \times {v_0}' \\
&\nabla_p \alpha(p, m) = {v_0}' \times {v_1}' + m' \times {v_1}' + {v_0}' \times m'.
\end{aligned} \tag{5.17}$$

Efficient Monte Carlo sampling of secondary edges is more involved. Unlike primary visibility where the viewpoint does not change much, shading point p can be anywhere in the scene. The consequence is that we need a more sophisticated data structure to prune the edges with zero contribution. Chapter 5.3 describes the process for importance sampling edges.

5.2.3 Cameras with Non-linear Projections

In Chapter 5.2.1, we desribed how we compute gradients for projective cameras. In this subsection, we discuss generalization of the camera model to handle non-linear projections. These cameras need a different treatment because we assumed we could obtain a closed-form of the edge equations in screen-space after projection. For projective cameras, a line in 3D is still a line after projection to screen space. For non-linear projections such as fisheye cameras, this assumption does not hold.

Fortunately, as Figure 5-6 illustrated, we can reuse the 3D extention we developed in the previous subsection for secondary visibility. For a camera location at position p and a 3D edge (v_0, v_1) in camera space, we can form the exact same edge equation $\alpha(p, m)$ as formulated in Equation 5.13. The gradient then can be computed the same way as in the previous section (Equation (5.16) and (5.17)).

5.2.4 Relation to Reynolds transport theorem and shape optimization

While we derive the gradients caused by discontinuities in the integrand using Dirac delta functions, it is also possible to represent the discontinuities using the integral boundary. Instead of using Heaviside functions to represent the shape boundaries, we can integrate over the domain on one side of the edge. Therefore, in the primary visibility case, Equation (5.6) can be rewritten as:

$$\nabla \iint \theta(\alpha_i(x,y)) f_i(x,y) dxdy = \nabla \iint_{\Omega_i(\Theta)} f_i(x,y) dxdy, \tag{5.18}$$

where $\Omega_i(\Theta)$ represents the domain where the Heaviside function $\theta(\alpha_i(x,y)) = 1$. Critically, the domain depends on the scene parameters Θ. Reynolds transport theorem [176], commonly used in fluid mechanics, addresses this specific case:

$$\begin{aligned} &\nabla_t \iint_{\Omega_i(\Theta)} f_i(x,y) dxdy = \\ &\iint_{\Omega_i(\Theta)} \nabla f_i(x,y) dxdy + \iint_{\partial\Omega_i(\Theta)} (v \cdot n)\, f_i(x(s), y(s)) ds, \end{aligned} \tag{5.19}$$

where $\partial\Omega_i(\Theta)$ is the boundary of the domain, v is the *velocity* at the boundary of the domain, and n is the normal vector of the boundary. This mirrors Equation (5.7) and also separates the gradient integral into a continuous part and a discontinuous part. If we choose to differentiate the x component, the velocity would be $(1, 0)$ and the normal before normalization would be $\left(a_y - b_y, b_x - a_x\right)$, which matches the results in Equation (5.9). Similar derivation also appears in shape optimization [200], where the goal is to find the optimal shape boundary that minimizes certain cost integral, using gradient-based optimization. Shape optimization has been applied in computer graphics in the context of diffusion curve optimization [241].

5.3 Importance Sampling the Edges

Our edge sampling method described in Chapter 5.2 requires us to sample an edge from hundreds of thousands, or even millions of triangles in the scene. The problem is two-fold:

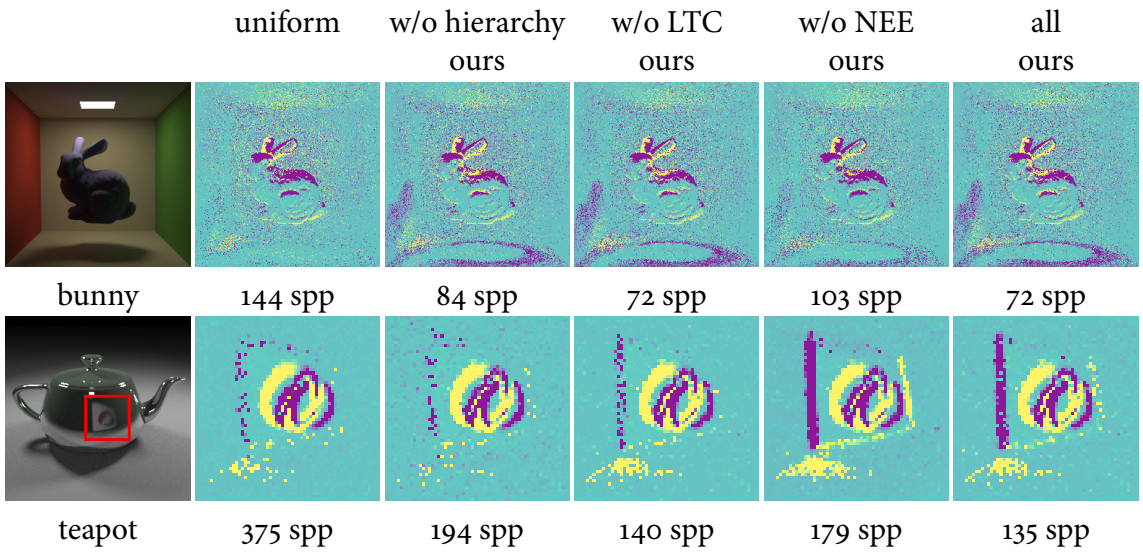


Figure 5-7: Equal time (24 seconds) comparison between sampling with and without our importance sampling methods. We tested our algorithm on scenes with soft shadow, global illumination, and specular reflection. We show the per-pixel derivatives of average color with respect to the bunny moving up in the top row, and the derivatives with respect to the reflected plane with the SIGGRAPH logo moving right in the second row. For the second row we only show the gradients in the red inset. *Uniform* indicates uniformly picking an edge based on length and uniformly picking a point on the edge, and is inefficient at sampling important contributions. We selectively turn off the three optimizations we introduce for importance sampling: the hierarchical edge selection (Chapter 5.3.1), the linearly transformed cosines (LTC) based importance sampling for a single edge (Chapter 5.3.2), and the sampling based on the next event estimation (NEE) rays' intersections with edge billboards (Chapter 5.3.3). Both the hierarchical edge selection and the LTC sampling are important for picking important or silhouette edges (see the teapot). The NEE intersection is important for shadow caused by relatively small light sources (see the bunny).

we need to sample an edge and then sample a point on the edge efficiently. Typically only a tiny fraction of these edges contribute to the gradients, since most of the edges are not silhouette (e.g. [83, 31]), and some of them are shadow blockers which can have significant contributions. Naive sampling methods fail to select important edges (Figure 5-7). Even if the number of edges is small, it is often the case that only a small region on an edge has non-zero contributions, especially when there exist highly-specular materials.

As mentioned in Chapter 5.2.1, the case for primary visibility is easier since the viewpoint is the camera. We project all edges onto the screen in a preprocessing pass, and test whether they are silhouettes with respect to the camera position. We sample an edge based on the distance of two points on the screen and uniformly sample in screen space. For secondary visibility, the problem is much more complicated. The viewpoint can be anywhere in the scene, and we need to take the material response between the viewpoint and the point on the edge into account.

In this section, we describe a scalable edge sampling implementation given arbitrary viewpoint. We introduce three optimizations: a hierarchical data structure for selecting important edges, an importance sampling scheme for selecting a point on a single edge, and a method that extracts shadow blocker edges using next event estimation rays. Our solution is inspired by previous methods for sampling many light sources using hierarchical data structures (e.g. [163, 221, 50]), efficient data structures for selecting silhouette edges [189, 84, 158], and the more recent closed-form solution for linear light sources [81, 82].

5.3.1 Edge selection

Given a shading point, our first task is to importance sample one or more triangle edges. There are several factors to take into account when selecting the edges: whether the edge is a silhouette, the geometric foreshortening factor inversely proportional to the distance to the edge, the material response between the shading point and the point on the edge, and the radiance incoming from the edge direction (e.g. whether it hits a light source or not). We address the incoming radiance in Chapter 5.3.3 and the rest in this subsection.

Our method involves building hierarchies of edges and probabilistically pruning out unimportance ones during traversal. We follow Olson and Zhang's Hough space approach [158] and build two hierarchies. The first contains the triangle edges that are silhouettes with respect to the camera position. The second contains the remaining edges. For the first set of edges, we build a 3D bounding volume hierarchy using the 3D positions of the two endpoints of an edge. For the second set of edges, we build a 6D bounding volume hierarchy using the two endpoint positions and the two planes associated with the two faces of an edge, transformed into the Hough space. For quick rejections of non-silhouette edges, we form a sphere between the shading point and the camera position (the *v-sphere* [158]), and test the intersection of the sphere and the bounding box of the planes in Hough space. We build the hierarchy parallelly by building a radix tree on top of the sorted Morton codes of the bounding box centroids [126, 107]. We also optimize for the surface area heuristics cost [142] using the treelet approach proposed by Karras and Aila [108].

We traverse the hierarchies to sample edges. For better stratification, we use a scheme similar to the Gaussian kd-tree [2]. We start with a number of samples at the root (say, 16). During the traversal, for each node in the hierarchy, we distribute the samples to the two children proportional to an importance estimation of the contributions, similar to the lightcuts algorithm [221]. We estimate the importance using the total length of edges, multiplied by the inverse distance to the center of the bounding box, times an upper bound estimation of the BRDF, by fitting the BRDF to a Linearly Transformed Cosine Distribution [81]. We

use the linearly transformed cosines for BRDF upper bound estimation since the linear transformation preserves the mode of the distribution, therefore we linearly transform the bounding box and compute the upper bound of the cosine distribution using Walter et al.'s method [221]. We set the importance to zero if the node does not contain any silhouette. We set the importance of both children to one if the shading point is inside the bounding box of their parent. Finally, we select one edge using weighted reservoir sampling [34], where the weight is the importance of the leaf node.

5.3.2 Importance sampling on an edge

Oftentimes with a highly-specular BRDF, only a small portion of the edges have significant contributions. We employ the recent technique on integrating linear light sources over the linearly transformed cosines [82]. Heitz and Hill's work provides a closed-form solution of the integral between a point and a linear light source, weighted by BRDF and geometric foreshortening. We numerically invert the integrated cumulative distribution function using Newton's method for importance sampling. We precompute a table of fitted linearly transformed cosines for our BRDFs.

5.3.3 Next event estimation for edges

The techniques above do not address the case of a small fraction of edges blocking a relatively small light sources. These edges cause sharp boundaries that have large gradients and are difficult to sample even with the edge hierarchy. We propose a method to address this by reusing the next event estimation samples we used in the forward pass. We transform all edges into billboards with finite width facing the next event estimation ray. We then collect all the edge billboards that intersect with the ray. We sample one of them based on the importance of the edge described in Chapter 5.3.1. We project the intersection of the ray and the billboard onto the nearest point on the edge, and the point on the edge is our sample. We set the width of the edge billboards to the L^2 sum of two percent of the mean absolute deviation of the triangle vertex positions over each coordinate.

We evaluate our importance sampling method using equal-time comparisons on GPU and show the results in Figure 5-7. We compare against the baseline approach of uniformly sampling all edges by length. We also perform ablation study by selectively turning off the three optimizations. The baseline approach is not able to efficiently sample rare events such as shadows cast by a small light source or highly-specular reflection of edges, while our importance sampling generates images with much lower variance.

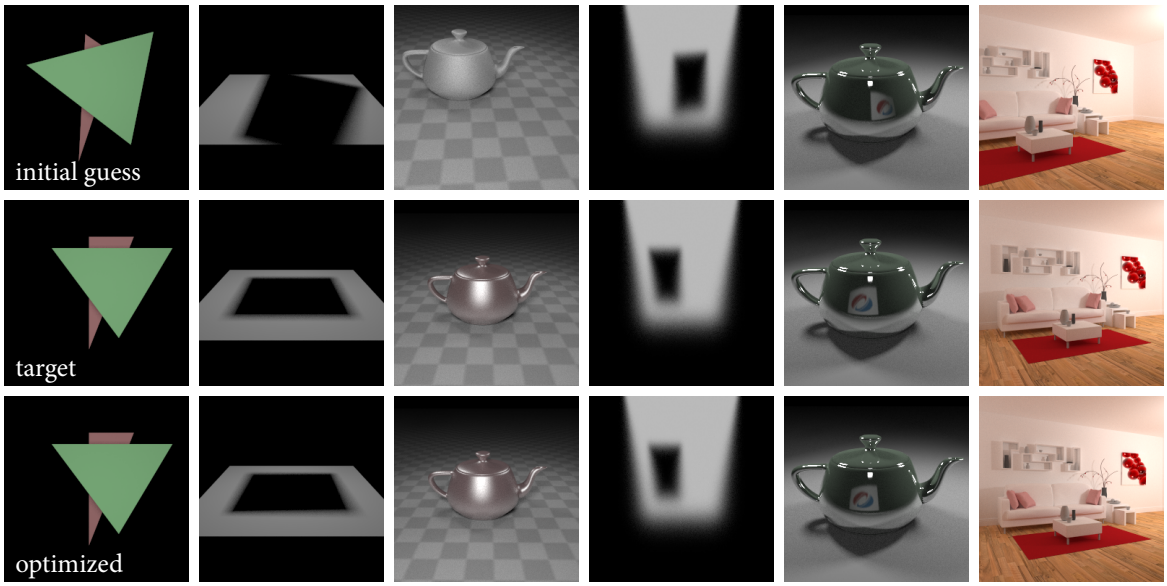
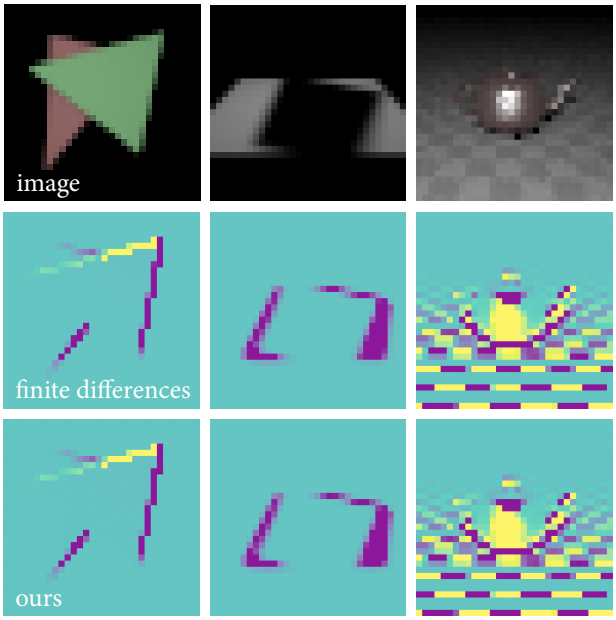


Figure 5-8: We verify our renderer by matching a range of synthetic scenes with different light transport configurations. For each scene, we start from an initial parameter (first row) and attempt to set scene parameters so that the rendering matches the target (second row) using gradient-based optimization. Each scene is intended to test a different aspect of the renderer. (a) optimizes triangle positions under the presence of occlusion. (b) optimizes blocker position for shadow. (c) optimizes camera pose and material parameters over textured and glossy surfaces. (d) optimizes the blocker position where the shadow receiver is highly glossy. (e) optimizes an almost specular reflection of a plane behind the camera; the free parameter is the plane position. (f) optimizes camera pose under the presence of global illumination and soft shadow. Our method is able to generate gradients for these scenes and to optimize the parameters correctly, resulting in minimal difference between the optimized result (final row) and target (second row). All the scenes are rendered with 4 samples per pixel during optimization. The final renderings are produced with 625 samples per pixel, except for (f) we use 4096 samples. We encourage the reader to refer to the project page (https://people.csail.mit.edu/tzumao/diffrt/) for videos and more scenes.

5.4 Results

We implement our method in a stand-alone C++/CUDA renderer with an interface to the automatic differentiation library PyTorch [165]. To use our system, the user constructs their scenes using lists of PyTorch tensors. For example, triangle vertices and indices are represented by floating point and integer tensors. Our renderer in the forward pass outputs an image which is also a PyTorch tensor. The user can then compute a scalar loss on the output image and obtain the gradient by backpropagating to the scene parameters.

Our renderer is structured similarly to a wavefront path tracer [123]. We trace one path per pixel at a time for all pixels. In the forward pass we store the path vertex information for



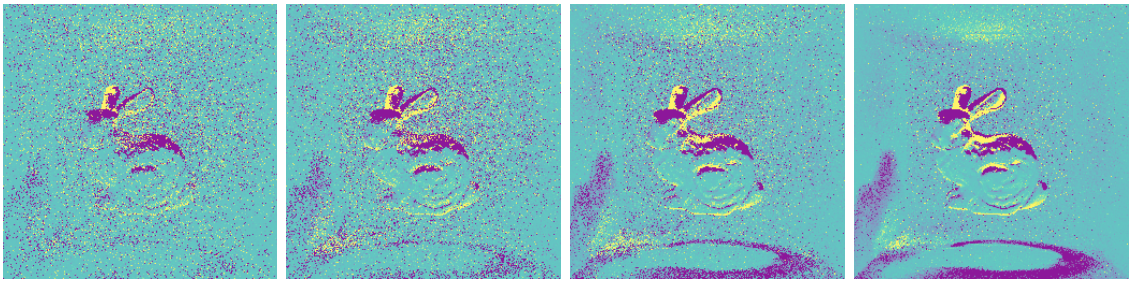
(a) triangles (b) shadow (c) teapot

Figure 5-9: We compare with central finite differences by rendering the scenes in Figure 5-8 at 32×32. The scenes are slightly adjusted to make the per-pixel gradient look clearer in the image. The derivatives are with respect to (a) each rightmost vertex of the two triangles moving left (b) the shadow blocker moving up (c) the camera moving into the screen. Our derivatives match the finite differences within an error of 1% relative to the L^1 norm of the gradients. Finite differences usually take two or three orders of magnitude more samples to reach the same error. For our method, we use 16 thousand samples per pixel for the scene with two triangles and 32 thousand samples per pixel for the other two scenes. For finite differences, we use 1 million samples per pixel for the triangles scene and 10 million samples per pixel for the rest.

each bounce (i.e. we checkpoint at each path vertex). We manually backpropagate both the edge gradients and smooth gradients in Equation (5.7) by traversing the light path backward (using in principle introduced in Chapter 2). We use Embree [220] and OptiX prime [164] for our ray casting operations. The renderer supports pinhole or thinlens camera with planar and equiangular spherical projection, Lambertian and Blinn-Phong BRDFs with Schlick approximation for Fresnel reflection, trilinear reconstruction of textures for diffuse and specular reflectance and roughness, area light sources with triangle meshes and environment maps.

5.4.1 Verification of the method

We tested our method on several synthetic scenes covering a variety of effects, including occlusion, non-Lambertian materials, and global illumination. Figure 5-8 shows the scenes. We start from an initial parameter, and optimize the parameters to minimize the L^2 differ-



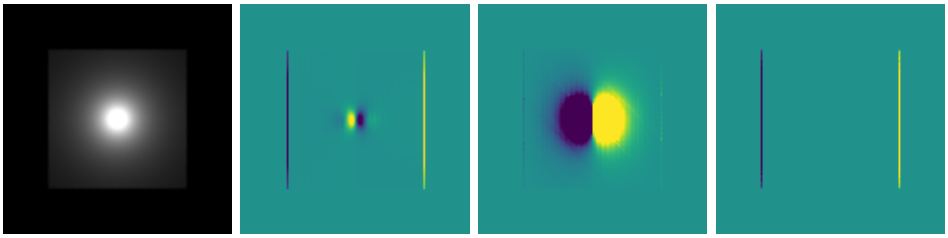
(a) 4 spp (b) 16 spp (c) 128 spp (d) 1024 spp

Figure 5-10: We visualize the per-pixel gradient contribution generated by our method over different numbers of samples per pixel. We take the bunny scene from Figure 5-7. The gradient is the average of color with respect to the bunny moving right. The 1024 samples per pixel image took around 5 minutes to compute on a Pascal GPU. In practice we usually use 4 samples per pixel for inverse rendering.

ence between the rendered image and target image using gradients generated by our method (except for the living room scene in Figure 5-8 (f) where we optimize for the L^2 difference between the Gaussian pyramids of the rendered image and target image). Our PyTorch interface allows us to apply their in-stock optimizers, and backpropagate to all scene parameters easily. We use the Adam [115] algorithm for optimization. The number of parameters ranges from 6 to 30. The experiment shows that our renderer is able to generate correct gradients for the optimizer to infer the scenes. It also shows that we are able to handle many different light transport scenarios, including cases where a triangle vertex is blocked but we still need to optimize it into the correct position, optimization of blocker position when we only see the shadow, joint optimization of camera and material parameters, pose estimation in presence of global illumination, optimizing blockers occluding highly-glossy reflection, and inverting near specular reflection. See the supplementary materials for more results.

We also compare our method to central finite differences on a lower resolution version of the synthetic scenes in Figure 5-9. Our derivatives match the finite difference within an error of 1% relative to the L^1 norm of the gradients. The comparison is roughly equal quality. We increase the number of samples for the finite differences until the error is low enough. In general finite differences require a small step size to measure the visibility gradient correctly, thus they usually take two or three orders of magnitude more samples to reach the same error as our result. In addition, finite differences do not scale with the number of parameters, making them impractical for most optimization tasks.

Figure 5-10 demonstrates the convergence of our method by visualizing the gradients of the bunny scene in Figure 5-7 over different numbers of samples per pixel. We show the gradients of the average of pixel colors with respect to the bunny moving right on the screen. Generating the near-converged 1024 samples per pixel image takes around 5 minutes on



(a) planar scene (b) OpenDR (c) Neural (d) ours

Figure 5-11: (a) A plane lit by a point light close to the plane. We are interested in the derivative of the image with respect to the plane moving right. Since the point light stays static, the derivatives should be zero except for the boundary. (b) (c) Previous work uses color buffer differences to approximate the derivatives, making them unable to take large variation between pixels into account and output non zero derivatives at the center. (d) Our method outputs the correct derivatives.

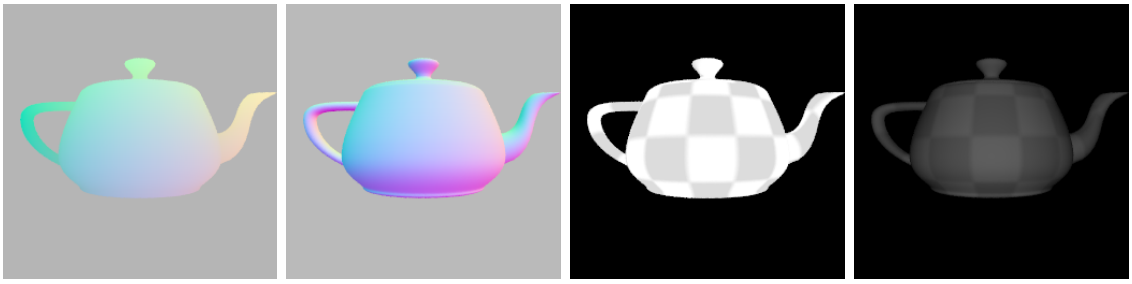
a Pascal GPU. In practice we don't render converged images for optimization. We utilize stochastic gradient descent and render a low sample count image (usually 4).

5.4.2 Comparison with previous differentiable renderers

In this subsection we compare our method with two previously proposed differentiable renderers: OpenDR [137] and Neural 3D Mesh renderer [109]. Both previous methods focus on speed and approximate the gradients even under Lambertian materials with unshadowed direct lighting. In contrast, our method outputs consistent gradients and supports arbitrary non-Dirac materials, shadow, and global illumination, as shown in Figure 5-8.

Both OpenDR and the Neural 3D Mesh renderer follow the approach of first rendering into a color buffer using a traditional rasterizer with z-buffer. They then approximate the derivatives with respect to screen-space triangle vertex positions using the rendered color buffer. OpenDR performs a screen-space filtering approach based on a brightness constancy assumption [103]. The shape of the filter is determined by boundary detection using triangle ID. For the horizontal derivatives of a pixel neighboring an occlusion boundary on the left, they use the kernel $[0, -1, 1]$. For pixels that are not neighboring any boundaries, or are intersecting with boundaries, or are neighboring more than one occlusion boundary, they use the kernel $\frac{1}{2}[-1, 0, 1]$. The Neural 3D Mesh renderer performs an extra edge rasterization pass of the triangle edges and accumulates the derivatives by computing the difference between the color difference on the color buffer around the edge. The derivative responses are modified by applying a smooth falloff.

Both previous differentiable renderers output incorrect gradients in the case where there is brightness variation between pixels due to lighting. Figure 5-11 shows an example of a plane lit by a point light with inverse squared distance falloff. We ask the two renderers and ours to



(a) 3D position (b) normal (c) albedo (d) deferred shading

Figure 5-12: **Geometry buffer rendering.** Our method can also output correct gradients of (a) 3D position (b) normal, (c). This enables vision applications such as matching RGB-D signals. The buffers can also be combined with deferred shading techniques to produce final rendering (d), for high-performance rendering.

compute the derivatives of the pixel color with respect to the plane moving right. Since the light source does not move, the illumination on the plane remains static and the derivatives should be zero except for the boundaries of the plane. Since both previous renderers use the differences between color buffer pixels to approximate derivatives, they incorrectly take the illumination variation as the changes that would happen if the plane moves right, and output non-zero derivatives around the highlights. On the other hand, since we sample *on* the edges, our method correctly outputs zero derivatives for continuous regions.

OpenDR's point light does not have distance falloff and the Neural 3D mesh renderer does not support point lights so we modified their renderers. Our renderer does not support pure point lights so we use a small planar area light to approximate a point light. We also tessellate the plane into 256×256 grids as both previous renderers use Gouraud shading.

5.4.3 Differentiable geometry buffer/AOV extension

Our method naturally generalizes to arbitrary shading functions, and can be used for generating geometry or AOV (arbitrary output variable) buffers, such as 3D position, normal, or material parameters. Our primary edge sampling (Chapter 5.2.1) backpropagates the derivatives of these auxiliary buffers correctly. This can be useful for computer vision applications (e.g. matching RGB-D signal), and also enables us to apply deferred shading techniques [44] to speed up rendering. Only coherent primary rays need to be traced in this case. Figure 5-12 shows the geometry buffers generated by our renderer.

5.4.4 Inverse rendering application

We apply our method on an inverse rendering task for fitting camera pose, material parameters, and light source intensity. Figure 5-1 shows the result. We take the scene photo and

geometry data from the thesis work of Jones [104], where the scene was used for validating daylight simulation. The scene contains strong indirect illumination and has non-Lambertian materials. We assign most of the materials to white except for plastic or metal-like objects, and choose an arbitrary camera pose as an initial guess. There are in total 177 parameters for this scene. We then use gradient-based optimizer Adam and the gradients generated by our method to find the correct camera pose and material/lighting parameters. In order to avoid getting stuck in local minima, we perform the optimization in a multi-scale fashion, starting from 64×64 and linearly increasing to the final resolution 512×512 through 8 stages. For each scale we use an L^1 loss and perform 50 iterations. We exclude the light source in the loss function by setting the weights of pixels with radiance larger than 5 to 0.

5.4.5 3D adversarial example

Recently, it has been shown that gradient-based optimization can also be used for finding adversarial examples for neural networks (e.g. [206, 64]) for analysis or mining training data. The idea is to take an image that was originally labelled correctly by a neural network classifier, and use backpropagation to find an image that minimizes the network's output with respect to the correct output. Our system can be used for mining adversarial examples of 3D scenes since it provides the ability to backpropagate from image to scene parameters. Similar ideas have been explored [10, 237, 135], but we use a more general renderer.

We demonstrate this in Figure 5-13. We show a stop sign classified correctly as a street sign by the VGG16 classifier [197]. We then optimize for 2256 parameters including camera pose, light intensity, sun position, global translation, rotation, and vertex displacement of the stop sign. We perform stochastic gradient descent to minimize the network's output of the classes street sign and traffic light, using 256 samples per pixel. After 5 iterations the network starts to output handrail as the most probable class. After 23 iterations both the street sign class and traffic light class are out of the top-5 predictions and the sum of the two has less than 5% probability.

We do not claim this as a robust way to break or to attack neural networks, since the CG scene we use has different statistics compared to real world images. Nevertheless this demonstrates that our gradient can be used for finding interesting scene configurations and can be potentially used for mining training data.

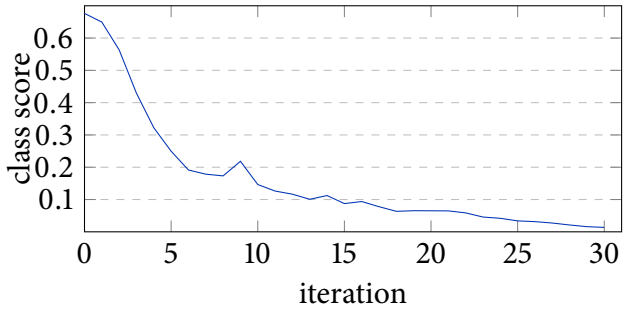
5.4.6 Limitations

Performance. Our current GPU implementation takes a few hundred milliseconds to generate a small resolution image (say 256×256) with a small number of samples (say 4). Note though

(a) input scene
53% street sign
14.5% traffic light
6.7% handrail

(b) 5 iterations
26.8% handrail
20.2% street sign
4.8% traffic light

(c) 25 iterations
23.3% handrail
3.39% street sign or
traffic light



(d) combined class score of street sign and traffic light

Figure 5-13: Our method can be used for finding 3D scenes as adversarial examples for neural networks. We use the gradient generated by our method to optimize for the geometry of the stop sign, camera pose, light intensity and direction to minimize the class scores of street sign and traffic light classes. After 5 iterations the network classifies the stop sign as a handrail, and after 25 iterations both street sign and traffic light are out of the top 5 predictions. In (d) we plot the sum of street sign and traffic light class scores as a function of iteration. As we optimize scene parameters such as the stop sign shape, gradient descent tries to find the geometry that minimizes the class scores, thus we see decreasing of the score.

that when using stochastic gradient descent it is usually not necessary to use high sample counts.

We have found that, depending on the type of scene, the bottleneck can be at the edge sampling phase or during automatic differentiation of the light paths, when we need to perform large reductions for the pixels hitting the same object (Chapter 4). Developing better sampling algorithms such as incorporating bidirectional path tracing [120] or photon mapping [100] could be an interesting avenue of future work. In particular, sampling the Dirac delta introduced by the edges is related to photon beams [98]. Developing better compiler techniques for optimizing automatic differentiation code is also an important task. While we achieved promising results in Chapter 4, the programming model focused on image processing and is unsuitable for many tasks in rendering such as tree traversal.

Other light transport phenomena. We assume static scenes with no participating media. Differentiating motion blur requires sampling on 4D edges with an extra time dimension. Combining our method with Gkioulekas et al.'s work [62] for handling participating media is left as future work.

Interpenetrating geometries and parallel edges. Dealing with the derivatives of interpenetration of triangles requires a mesh splitting process and its derivatives. Interpeneration can happen if the mesh is generated by some simulation process. As shown in Figure 5-4, our method also does not handle the case where two edges are perfectly aligned as seen from the center of projection (camera or shadow ray origin). However, these are zero-measure sets in path space, and as long as the two edges are not perfectly aligned to the viewport, we will be able to converge to the correct solution.

Shader discontinuities. We assume our BRDF models and shaders are differentiable and do not handle discontinuities in the shaders. We handle textures correctly by differentiating through the smooth reconstruction, and many widely-used reflection models such as GGX [222] (with Smith masking) or Disney's principled BRDF [29] are differentiable. However, we do not handle the discontinuities at total internal reflection and some other BRDFs relying on discrete operations, such as the discrete stochastic microfacet model of Jakob et al. [96]. Extremely high frequency textures also require prefiltering to have low variance on both the rendered images and the gradients. Compiler techniques for band-limiting BRDFs can be applied to mitigate the shader discontinuity issue [232].

5.5 Conclusion

We have introduced a differentiable Monte Carlo ray tracing algorithm that is capable of generating correct and unbiased gradients with respect to arbitrary input parameters such as scene geometry, camera, lights and materials. For this, we have introduced a novel edge sampling algorithm to take the geometric discontinuities into consideration, and derived the appropriate measure conversion. For increased efficiency, we use a new discrete sampling method to focus on relevant edges as well as continuous edge importance sampling. We believe this method and the software that we release will have an impact in inverse rendering and deep learning.

5.A Derivation of the 3D edge Jacobian

We derive the Jacobian $J_m(t)$ in Equation 5.16. The goal is to compute the derivatives of point $m(t)$ with respect to the line parameter t. The relation between $m(t)$ and t is described by a

ray-plane intersection. That is, we are intersecting a plane at point m with normal n_m with a ray of origin p and unnormalized direction $\omega(t)$:

$$\begin{aligned} \omega(t) &= v_0 + (v_1 - v_0)t - p \\ \tau(t) &= \frac{(m-p)\cdot n_m}{\omega(t)\cdot n_m} \\ m(t) &= \tau(t)\omega(t). \end{aligned} \tag{5.20}$$

We can then derive the derivative $J_m(t) = \frac{\partial m(t)}{\partial t}$ as:

$$J_m(t) = \tau(t)\left((v_1 - v_0) - \omega(t)\frac{(v_1 - v_0)\cdot n_m}{\omega(t)\cdot n_m}\right) \tag{5.21}$$

6 | Hessian-Hamiltonian Monte Carlo Rendering

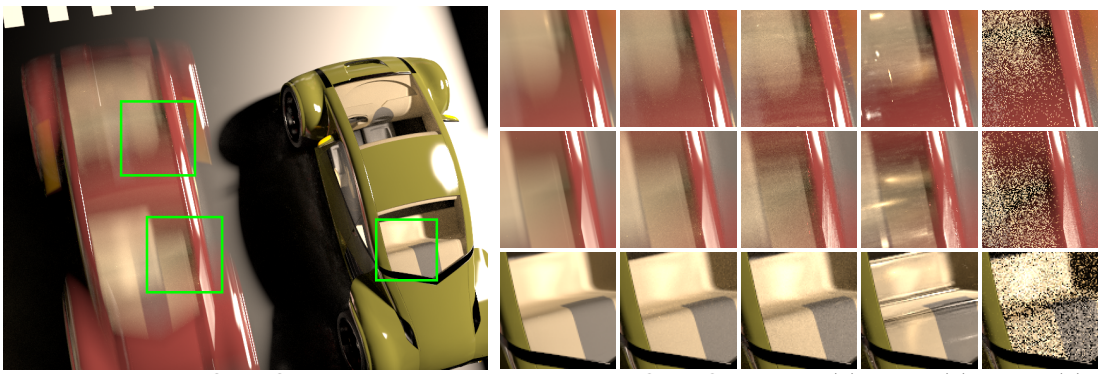


Figure 6-1: CARS: Equal-time (20 minutes) comparison on the *cars* scene, with a static car and a moving car lit by an area light. The direct lighting is computed separately. The interior of the car is enclosed by near-specular glass windows, which gives rise to specular-diffuse-specular paths that are challenging to sample. The three insets show the renderings of our method (H^2MC), Manifold Exploration Metropolis Light Transport (MEMLT) [97], Multiplexed Metropolis Light Transport (MMLT) [74], and Bidirectional Path Tracing (BDPT) [214]. The reference (REF) is rendered by our method in roughly 15 hours. BDPT cannot efficiently sample the sparse contribution function and suffers from severe noise. MMLT tends to get trapped in the hard-to-find features and produces correlated noise. MEMLT specializes in finding difficult static specular paths, but does not consider the anisotropy in the time domain, resulting in ghosting artifacts. Our method can efficiently resolve the hard-to-find caustics light paths like the specialized method, and is more general so that it can resolve moving caustic paths in the window by capturing the correlation between the time domain and path space.

In the previous chapters, we focused on *inverse* applications, where we try to find a set of parameters or inputs satisfying certain outputs. In this chapter, we discuss something slightly different, in particular we show that derivatives can also be useful for accelerating *forward* rendering.

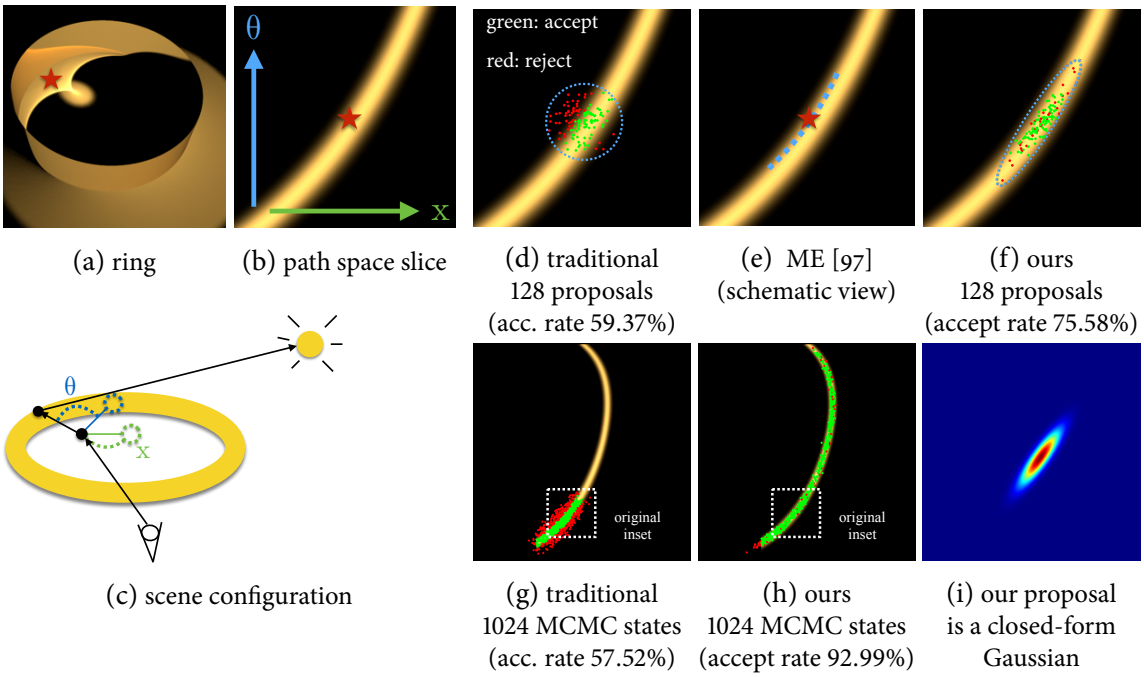
Light transport phenomena such as caustics, multiple-bounce glossy transport and motion blur often concentrate high contributions in a narrow volume within the high-dimensional sample space. While efficient methods exist for local importance sampling of individual

scattering events, their combined effect on path throughput is intricate and hard to sample, leading to noisy images. Figure 6-2 shows a caustic caused by a glossy gold ring. The integrand (Figure 6-2 (b)) is sparse: for points on the floor (x), only a few incident directions (θ) contribute radiance through reflection. Even in this simple scene, sparsity makes standard numerical integration methods inefficient. The region of high-contribution is continuous, but highly anisotropic, and the anisotropy varies over the integrand. In this chapter, we present a general solution by extending Metropolis Light Transport [216] (a Markov Chain Monte Carlo sampler) to exploit the local structure of the path contribution function over its entire high-dimensional domain.

Adapting to the local anisotropic behavior of the integrand has been a long-standing challenge in rendering. Previous work has focused on model-based characterizations of anisotropy that are tied to specific effects (specular transfer, motion, etc.) [97, 15, 106], and combining them is not easy. Closest to our work is Manifold Exploration [97] and Half-vector Space Light Transport [106, 75] which use assumptions about the mirror direction and specular reflection to derive major directions of anisotropy (Figure 6-2e), and walk along a lower-dimensional manifold. In contrast, we seek for a general solution that can characterize the "thickness" of the manifold in all directions, avoiding case-specific manual derivations.

The adaptation boils down to two main problems: 1) characterizing the anisotropy using local information and 2) sampling according to the derived information. We solve 1) by characterizing the local throughput using its derivatives. Since the gradient provides weak directional information, we also use the second derivative, the Hessian matrix. Whereas the gradient points only into the direction of the strongest increase, the Hessian additionally captures the correlation between coordinates. While the Hessian has been used before in rendering, e.g. [87, 193], its manual derivation is tedious and has usually been restricted to specific transport phenomena such as diffuse-only. In contrast, we use automatic differentiation (Chapter 2) which allows us to handle general effects.

While the Hessian captures anisotropy well, the second problem of sampling remains: it is not possible to directly sample from the resulting quadratic approximation because it does not define a proper distribution and grows to infinity. Instead, we start from Hamiltonian Monte Carlo [48], a Markov chain Monte Carlo sampling algorithm that proposes new sample locations by simulating the dynamics of a particle that starts at the current sample with a random initial velocity, briefly mentioned in Chapter 3.2.3. The particle evolves under gravity in a landscape composed of the contribution function flipped upside down so that the particle is attracted to high contribution areas (low height) by gravity. Crucially, we do not apply Hamiltonian Monte Carlo directly: this would be too expensive, because it would require numerical integration to generate just a single sample, and each integration time step



(a) ring (b) path space slice (d) traditional 128 proposals (acc. rate 59.37%) (e) ME [97] (schematic view) (f) ours 128 proposals (accept rate 75.58%)

(c) scene configuration (g) traditional 1024 MCMC states (acc. rate 57.52%) (h) ours 1024 MCMC states (accept rate 92.99%) (i) our proposal is a closed-form Gaussian

Figure 6-2: RING EXAMPLE: (a) A motivating example showing the caustics caused by a highly-glossy gold ring, lit by a distant point light. (b) A slice of the two-bounce indirect light field around the red star, where x represents one of the dimensions in screen-space, and θ represents one of the dimensions along the BRDF sampling direction (the configuration is shown in (c)). The path contribution is sparse, and most of the contributions are zero. (d) The green/red dots represent the accepted/rejected proposal samples of a traditional MCMC rendering algorithm [111], which uses isotropic mutation that makes the sampling inefficient. (e) We also show the schematic of Manifold Exploration (ME) [97], which only travels on the tangent of a lower dimensional space. (f) Our approach builds a Gaussian approximation around the neighborhood, enabling us to efficiently traverse the target function locally. Some samples are rejected due to the adaptivity, but it still results in a higher acceptance rate. (g)(h) We show the zoomed out slice (the positions of the original insets (d) and (e) are at the bottom of the images) with dots now representing the samples obtained by simulating the Markov chain for 1024 states; our method explores the space more thoroughly. We also show the false color visualization of the Gaussian approximation in (i), which takes the width of the function into consideration.

would involve costly ray tracing, shading, and derivatives which do not directly contribute to the image. In practice, up to a hundred time steps per sample may be needed [155]. Instead, we apply a modified version of HMC that results in closed-form integration. As we show in the paper, running Hamiltonian dynamics on a second-order function with a Gaussian distribution of initial momentums leads to a Gaussian distribution of final positions, and it results in a standard Metropolis-Hastings sampling. While traditional Metropolis sampling also uses a Gaussian distribution of proposals, it is usually isotropic and is centered on the current sample. In contrast, our Gaussian proposal is anisotropic, conforms to the shape of the contribution function, and is centered towards higher values according to the local gradient and Hessian.

While our method can be used with arbitrary path parametrization, a carefully designed parametrization can lead to better mixing of the Markov chain. We propose a modified parameterization of the path space based on the *primary sample space* proposed by Kelemen et al. [111]. The modified parameterization reduces the correlation between the dimensions (Figure 6-7).

Our method is general thanks to the use of the second-order Taylor expansion and automatic differentiation. In particular, it can be easily extended to time for motion blur effects, so that we are able to resolve the correlation between path-space and time for a light path that contains a moving caustic in a window (Figure 6-1). We focus on surface rendering in this paper, though conceptually our general approach could be extended to handle a variety of other phenomena such as BSSRDFs or participating media.

6.1 Related Work

Our work is closely related to the rendering algorithms that build upon MCMC sampling and the methods that utilize derivatives to drive the sampling process.

Metropolis Light Transport In light transport simulation, we need to compute the path integral [213] I_j for each pixel j:

$$I_j = \int_\Omega h_j(\boldsymbol{x})\, f(\boldsymbol{x})\, d\mu(\boldsymbol{x}), \tag{6.1}$$

where Ω is *path space*, which contains all the light paths, h_j is the camera response function for pixel j, $f(\boldsymbol{x})$ is the path contribution function [213], and $\mu(\boldsymbol{x})$ is the area density of path $\boldsymbol{x}$.

Veach and Guibas [216] apply the Markov Chain Monte Carlo sampling method (see Chapter 3.2 for a brief review) by generating a sequence of Markov chain samples $\boldsymbol{x}_i$. A new proposal sample is *mutated* from the previous sample, and probabilistically accepted or rejected. Specifically, given a sample $\boldsymbol{x}_{i-1}$, and a target function $f^*(\boldsymbol{x})$, which is commonly set to the luminance of $f(\boldsymbol{x})$, we first generate a *proposal* sample $\boldsymbol{x}'$ with the transition probability $Q(\boldsymbol{x}_{i-1} \to \boldsymbol{x}')$, and set the next sample $\boldsymbol{x}_i$ as follows:

$$\boldsymbol{x}_i = \begin{cases} \boldsymbol{x}' & \text{with probability } a(\boldsymbol{x}_{i-1} \to \boldsymbol{x}') \\ \boldsymbol{x}_{i-1} & \text{otherwise,} \end{cases} \tag{6.2}$$

where the acceptance probability a is defined as

$$a\left(\boldsymbol{x}_{i-1} \rightarrow \boldsymbol{x}'\right) = \min\left(1, \frac{f^*\left(\boldsymbol{x}'\right) Q\left(\boldsymbol{x}' \rightarrow \boldsymbol{x}_{i-1}\right)}{f^*\left(\boldsymbol{x}_{i-1}\right) Q\left(\boldsymbol{x}_{i-1} \rightarrow \boldsymbol{x}'\right)}\right) \tag{6.3}$$

This satisfies the *detailed balance* condition. That is, for any light paths $\boldsymbol{x}$ and $\boldsymbol{y}$, we have

$$f^*\left(\boldsymbol{x}\right) Q\left(\boldsymbol{x} \rightarrow \boldsymbol{y}\right) a\left(\boldsymbol{x} \rightarrow \boldsymbol{y}\right) = f^*\left(\boldsymbol{y}\right) Q\left(\boldsymbol{y} \rightarrow \boldsymbol{x}\right) a\left(\boldsymbol{y} \rightarrow \boldsymbol{x}\right). \tag{6.4}$$

As mentioned in Chapter 3.2, if a transition function satisfies the detailed balance condition, and if there is a strict positive probability to sample all light paths with non-zero contribution (*ergodicity*), it will converge to a distribution proportional to the target function $f^*(\boldsymbol{x})$. Veach and Guibas then approximated the path integral I_j at pixel j using the weighted average of the Markov chain samples:

$$I_j = \frac{b}{N} \sum_{i=1}^{N} \frac{h_j\left(\boldsymbol{x}_i\right) f\left(\boldsymbol{x}\right)}{f^*\left(\boldsymbol{x}\right)}, \tag{6.5}$$

where b is a normalization constant, which is the average of $f^*\left(\boldsymbol{x}\right)$ over the image.[1] Originally Veach and Guibas designed several specialized mutation strategies to address different lighting scenarios. Each strategy has a different asymmetric probability distribution, which introduces a significant challenge to implement all the strategies correctly. To simplify the algorithm, Kelemen et al. [111] proposed to mutate the state in the *random number space*, which makes the mutation agnostic to the particular lighting effect. Later, Jacopo [162], Otsu et al. [23] and Bitter et al. [23] propose methods to combine the two mutation strategies by finding the random number that generates a certain light path produced by Veach and Guibas's mutation. Unfortunately, both the mutation strategies proposed by Veach and Guibas and Kelemen et al. do not respect the complex local structure in the sampling domain, which makes them inefficient in some difficult cases.

Metropolis Light Transport has been extended in several aspects. Cline et al. [42] proposed the Energy Redistribution Path Tracing technique by running many short Markov chains. Lai et al. [122] adapted mutations with different parameters using Population Monte Carlo. Kitaoka et al. [116] introduced replica exchange, or parallel tempering [205], that exchanges states between multiple Markov chains to avoid getting stuck at local modes. All these methods require some form of local mutation strategies. We introduce a new local sampling strategy that adapts to the local structure of the function. Lai et al. [121] proposed a temporal

[1] Since we need to estimate the normalizing constant b, this particular use of Markov chain is only useful when we are interested in multiple integrals, where their integrals are correlated.

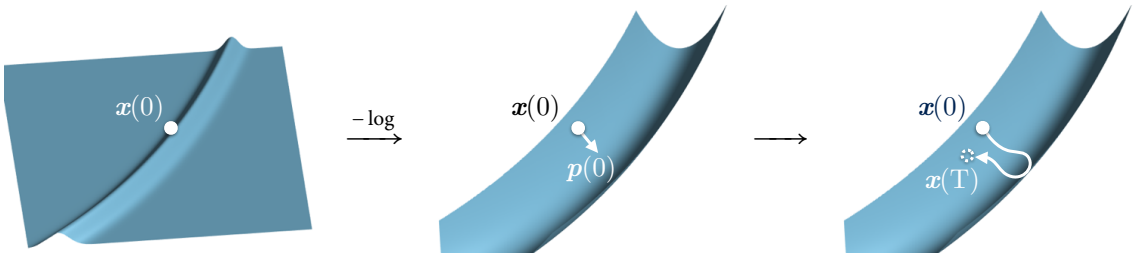
mutation strategy based on object-space transformation. Unlike their method, which requires a specially designed mutation, we treat the time dimension the same as the other dimensions, and handle the correlation between coordinates using second derivatives.

Jakob and Marschner [97], Kaplanyan et al. [106], and Hanika et al. [75] use the first derivatives of the half-vectors of a specular light path to guide the MCMC sampling. These methods apply a form of Newton-iteration to sample new light paths satisfying certain constraints. While they improve the sampling efficiency of glossy and specular surfaces significantly, their methods can sometimes be inefficient on small, highly-curved surfaces, because of their first-order approximation. In addition, they only account for a subset of terms in the path contribution function, ignoring important effects such as the Fresnel reflection or light source emission profiles. In contrast, we utilize second-order derivatives and do not assume any particular effect. For example, we are able to render difficult moving caustics (Figure 6-1), where their methods would suffer from ghosting artifacts

Hachisuka et al. [74] proposed Multiplexed Metropolis Light Transport that combines Kelemen et al's mutation startegy with multiple importance sampling [215]. Their method is orthogonal to our algorithm, and we build our bidirectional path tracer based on their approach.

Derivatives in rendering Shinya et al. [196] used a second-order power series along with paraxial approximation to approximate the neighborhood of a ray. Irradiance caching techniques [225, 224, 193] compute the gradients and the Hessians of the irradiance with respect to the screen coordinates for sparse interpolation for diffuse or low-glossy surfaces. Ray differentials [90] and path differentials [204] compute the footprint of the light paths for texture filtering using first derivatives. Chen and Arvo [37] use first and second-order derivatives of the specular light paths for sparse interpolation. Path gradients [204] are used for hierarchical radiosity applications, where the gradients of the paths are hand-derived. Ramamoorthi et al. [173] performed a first-order analysis for the direct illumination light field. Gradient-domain rendering approaches, e.g. [127, 113], sample in the gradient domain to exploit the sparsity of gradients in image space. They use finite differences of the path on the image coordinates, whereas our method uses analytical derivatives of all dimensions. While finite differences could capture the discontinuities of the signal, they are more expensive to generate and do not scale well with dimensionality.

Our usage of derivatives differs from previous works in several respects. First, we use automatic differentiation to compute the derivatives, which means that we do not assume any particular effect. This enables our method to handle various combinations of lighting scenarios. Second, we take the derivatives with respect to all the sampling dimensions, so



(a) given current sample $\boldsymbol{x}(0)$ and target function

(b) flip function so that gravity pulls the particle, sample initial momentum $\boldsymbol{p}(0)$

(c) simulate Hamiltonian dynamics for time T to obtain proposal at $\boldsymbol{x}(T)$

Figure 6-3: HAMILTONIAN MONTE CARLO: Given the current sample position $\boldsymbol{x}(0)$ and a target function (the 2D slice from Figure 6-2), a physical analogy of Hamiltonian Monte Carlo is: (a) first it takes the logarithm of the target function and flips it upside down so that "gravity" pulls towards high contribution areas. (b) Then it gives the current sample an initial momentum $\boldsymbol{p}(0)$ and (c) lets the point move for some time T with respect to the geometry of the flipped function.

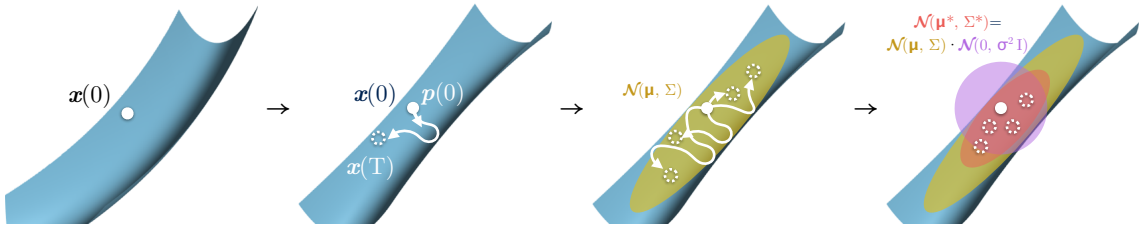
we can capture the high-dimensional structure of the light path. Third, we take both the first and the second derivatives. The Hessians enable us to take the correlation between the dimensions of the sampling domain into account. Finally, we apply the derivatives in the MCMC sampling context.

6.2 Hamiltonian Monte Carlo

We review the Hamiltonian Monte Carlo [48] method; also see Neal's introduction article [155] for a more thorough description and survey. Hamiltonian Monte Carlo generates the new proposal samples from the current sample by simulating Hamiltonian dynamics driven by the landscape of the target function.

Markov chain Monte Carlo methods generate a sequence of samples $\boldsymbol{x}_i$, whose distribution converges to a distribution proportional to a specific target function $f^*(\boldsymbol{x}_i)$, by forming a Markov chain of the sample sequence. For the sake of notational simplicity, we denote the target function as $f(\boldsymbol{x})$. At iteration $i+1$, a new proposal sample is drawn from a distribution based on $\boldsymbol{x}_i$. Then the proposal sample is probabilistically accepted or rejected. If accepted, it forms the new state of the Markov chain. From now on, we assume that the samples $\boldsymbol{x}_i$ lie in a hypercube of $[0,1]^N$, similar to the primary sample space [111]. Operating directly in path-space [213] is more challenging due to its definition as a cross product of lower-dimensional manifolds.

Figure 6-3 gives an illustration of Hamiltonian Monte Carlo where, in a nutshell, state is modified by giving the current sample a random initial velocity (or more precisely, a momentum), and simulating its motion under gravity. The target function first needs to



(a) original target and sample position

(b) simulate dynamics on local quadratic approximation

(c) proposals result in a Gaussian distribution

(d) multiply with a prior to limit variance, draw from red

Figure 6-4: Hessian-Hamiltonian Monte Carlo: (a) Given the original function and sample position $x\,(0)$, (b) we approximate the costly Hamiltonian dynamics simulation by first constructing a local quadratic approximation at the current sample position $\boldsymbol{x}\,(0)$. (c) Different initial momentum $\boldsymbol{p}\,(0)$ results in different proposal positions $\boldsymbol{x}\,(T)$, which makes $\boldsymbol{x}\,(T)$ a random variable. We show that trajectories with a Gaussian distribution for initial momentum result in a Gaussian distribution (the yellow shaded area) for final destination. (d) Finally, we multiply the Gaussian with a prior (the purple shaded area) to prevent proposals from going too far when the second derivative is low. The resulting sample proposal distribution is shown in red, and we draw our proposals from the resulting distribution.

be "flipped" so that high contribution regions correspond to a lower height and samples are attracted there by gravity (Figure 6-3 (b)). The particle is given an initial momentum, typically drawn from a Gaussian, and its motion is simulated in the height field given by the flipped contribution function for a fixed amount of time. Acceptance rules are then applied, although if the integrator preserves energy, samples are always accepted. This approach helps the samples stay in the high contribution region (low height in the flipped function) because of the effect of gravity.

Hamiltonian dynamics Formally, Hamiltonian dynamics is a system of differential equations defined on the Hamiltonian energy E:

$$\begin{aligned} \frac{\partial \boldsymbol{x}}{\partial t} &= \frac{\partial E}{\partial \boldsymbol{p}} \\ \frac{\partial \boldsymbol{p}}{\partial t} &= -\frac{\partial E}{\partial \boldsymbol{x}}. \end{aligned} \tag{6.6}$$

The auxiliary momentum variable $\boldsymbol{p}$ is introduced to drive the sampling of position $\boldsymbol{x}$, and $\boldsymbol{p}$ has the same number of dimensions as $\boldsymbol{x}$. The Hamiltonian energy $E\,(\boldsymbol{x}, \boldsymbol{p})$ is a composite of the potential energy $U\,(\boldsymbol{x}) = -\log f\,(\boldsymbol{x})$ and the kinetic energy $K\,(\boldsymbol{p}) = \frac{1}{2}\boldsymbol{p}^T A \boldsymbol{p}$. The potential energy is defined in the logarithmic domain to better capture the dynamic range of

the target functions:

$$E(\boldsymbol{x}, \boldsymbol{p}) = U(\boldsymbol{x}) + K(\boldsymbol{p}) = -\log f(\boldsymbol{x}) + \frac{1}{2}\boldsymbol{p}^T A \boldsymbol{p}, \tag{6.7}$$

where A is a user-defined "inverse mass matrix", which represents the inverse of the mass of the particle. Typically, it is set to a scalar $\frac{1}{m}$ times an identity matrix, where m is the mass, but in our work we will use a full matrix (discussed in Section 6.3). The negative of the function $\log f(x)$ is taken to enable high contribution regions to have low potential energy (as shown in Figure 6-3).

We substitute the definition of the Hamiltonian energy (Equation (6.7)) into the Hamiltonian equation (Equation (6.6)), and obtain:

$$\begin{aligned} \frac{\partial \boldsymbol{x}}{\partial t} &= A\boldsymbol{p} \\ \frac{\partial \boldsymbol{p}}{\partial t} &= \frac{\partial \log f(\boldsymbol{x})}{\partial \boldsymbol{x}}. \end{aligned} \tag{6.8}$$

Equation (6.8) defines a trajectory of position $\boldsymbol{x}$ and momentum $\boldsymbol{p}$ over time t. Intuitively, if the momentum at time t is high, we will make a large jump from the current position $\boldsymbol{x}(t)$, and if the derivatives of the target function at $\boldsymbol{x}(t)$ are low, the increment to the momentum will be small. Hamiltonian Monte Carlo is highly adaptive to the local structure of the target function.

Markov chain Monte Carlo with Hamiltonian dynamics To apply Hamiltonian dynamics in the context of Markov chain Monte Carlo, we first take the exponent of the negative Hamiltonian energy:

$$\exp(-E(\boldsymbol{x}, \boldsymbol{p})) = f(\boldsymbol{x}) \exp\left(-\frac{1}{2}\boldsymbol{p}^T A \boldsymbol{p}\right) =: f(\boldsymbol{x})\phi(\boldsymbol{p}). \tag{6.9}$$

$\exp\left(-\frac{1}{2}\boldsymbol{p}^T A \boldsymbol{p}\right)$, which we denote as $\phi(\boldsymbol{p})$, is proportional to the PDF of a zero-mean Gaussian with covariance A^{-1}. To generate a new proposal position, we pick a zero-mean Gaussian distributed momentum $\boldsymbol{p}(0)$ with covariance A^{-1} and a fixed time T, and simulate the Hamiltonian dynamics to obtain the position at $\boldsymbol{x}(T)$.

The proposal position is probabilistically accepted with the probability a, where

$$\begin{aligned} & a\left(\left(\boldsymbol{x}\left(0\right),\boldsymbol{p}\left(0\right)\right)\rightarrow\left(\boldsymbol{x}\left(T\right),\boldsymbol{p}\left(T\right)\right)\right) \\ & \quad=\min\left(\frac{\exp\left(-E\left(\boldsymbol{x}\left(T\right),\boldsymbol{p}\left(T\right)\right)\right)}{\exp\left(-E\left(\boldsymbol{x}\left(0\right),\boldsymbol{p}\left(0\right)\right)\right)},1\right) \\ & \quad=\min\left(\frac{f\left(\boldsymbol{x}\left(T\right)\right)\phi\left(\boldsymbol{p}\left(T\right)\right)}{f\left(\boldsymbol{x}\left(0\right)\right)\phi\left(\boldsymbol{p}\left(0\right)\right)},1\right). \end{aligned} \tag{6.10}$$

Intuitively, the acceptance rule resembles the Metropolis-Hastings rule (Equations (6.2) and (6.3)), where the transition probability T is substituted with the (unnormalized) PDF of the momentum Gaussian ϕ. Furthermore, if the Hamiltonian dynamics is simulated perfectly, $\exp\left(-E\left(\boldsymbol{x},\boldsymbol{p}\right)\right)$ is a constant throughout the simulation because of energy conservation, and the acceptance probability is 1.

Properties of Hamiltonian dynamics More formally, given a fixed time T, the Hamiltonian equation creates a mapping M between $\left(\boldsymbol{x}\left(0\right),\boldsymbol{p}\left(0\right)\right)$ and $\left(\boldsymbol{x}\left(T\right),\boldsymbol{p}\left(T\right)\right)$. Neal [155] showed that this mapping has several important properties:

1. The mapping is time-reversible: if we flip the momentum at time T and use $\left(\boldsymbol{x}\left(T\right),-\boldsymbol{p}\left(T\right)\right)$ as the input to M, the output of the mapping would be $\left(\boldsymbol{x}\left(0\right),-\boldsymbol{p}\left(0\right)\right)$. That is, if we simulate the Hamiltonian dynamics in a backward manner from the end point, it will go back to the starting point.

2. The mapping preserves the volume: If we apply the mapping for a region R_0 of points $\left(\boldsymbol{x}\left(0\right),\boldsymbol{p}\left(0\right)\right)$, and map them to another region R_T, the volumes of the two regions in the position-momentum space remain the same (known as Liouville's theorem).

3. The mapping preserves energy: the Hamiltonian energy E (Equation (6.7)) remains the same after the mapping.

The first property is crucial for the detailed balance condition (Equation (6.4)) to hold, since it ensures that the mapping is one-to-one. The second property ensures that we do not need to account for the Jacobian of the mapping in the Metropolis acceptance rule. The energy preservation property shows that the probability of acceptance is in fact 1 since $E\left(\boldsymbol{x}\left(0\right),\boldsymbol{p}\left(0\right)\right)=E\left(\boldsymbol{x}\left(T\right),\boldsymbol{p}\left(T\right)\right)$. Recently, it has been shown [199] that it is also possible to design transition rules for Hamiltonian Monte Carlo to converge without satisfying the detailed balance condition.

Unfortunately, Equation (6.8) does not have a known analytical solution for an arbitrary target function. It is usually required to integrate the differential equation using numerical

integrators such as leapfrog integrators. These integrators maintain the time-reversibility and volume-preservation, but do not preserve energy. The Hamiltonian dynamics are approximated and the acceptance probability is no longer 1. Furthermore, numerical integrators are expensive for light transport simulation because each step involves costly ray tracing operations and derivative computations of the shader.

Discussion. As discussed in Chapter 3.2, Langevin Monte Carlo [181] is a one-step approximation to Hamiltonian Monte Carlo. Its proposal distribution is isotropic, except that the mean of the proposal distribution is shifted by the first derivatives (gradient) times a user-specified constant. Our method is also a one-step approximation, but the proposal distribution of our method adapts to the anisotropy of the signal, because we utilize the second derivatives. It is possible to precondition Langevin Monte Carlo using a positive-definite mass matrix, such as the Fisher information matrix [60]. However, it remains unclear how to relate the Hessian matrix to the positive-definite mass matrix. Betancourt [18] proposed the SoftAbs metric that removes the sign of the eigenvalues of the Hessian matrix using a smooth mapping. In contrast, we treat positive and negative eigenvalues differently by directly simulating Hamiltonian dynamics on the quadratic landscape.

6.3 Hessian-Hamiltonian Monte Carlo

Figure 6-4 illustrates our sampling algorithm. We compute the second order Taylor expansion (local quadratic approximation) of the logarithm of the target function first, where the gradient and the Hessian are computed using automatic differentiation. The quadratic function does not define a proper distribution, since it might grow to infinity, which prevents us from directly importance sampling it. Hamiltonian dynamics enables us to sample from this quadratic function to obtain the proposal position, since it works on any continuous function.

The Hamiltonian dynamics have an analytical solution in the case of a quadratic function. However, we cannot use the acceptance rule in standard Hamiltonian Monte Carlo (Equation (6.10)) to compute the acceptance probability. It would break time-reversibility, since each light path would have a different associated quadratic function. Fortunately, we can derive from the analytical solution that the distribution of a proposal, given a Gaussian momentum, is a Gaussian distribution (Figure 6-4 (c)). Therefore, we associate each light path with a Gaussian distribution derived from the quadratic function and Hamiltonian dynamics, and it is possible to compute the acceptance probability using the Metropolis-Hastings rule (Equation (6.3)). Finally, we multiply the analytical Gaussian with a prior

Gaussian distribution to place a limit on its variance (Figure 6-4 (d)), so that the proposals do not go too far away where the second order approximation can be inaccurate.

Approximating Hamiltonian dynamics We first show how to derive the closed-form solution to the differential equations for Hamiltonian dynamics (Equation (6.8)), given an initial momentum and position. Then, we will show how to infer the Gaussian distribution of proposals. We start from a second-order approximation of $\log f$. For the sake of simplicity and without loss of generality, in the following we assume the current position $\boldsymbol{x}(0)$ is at the origin. Any small offset $\boldsymbol{x}$ from the origin can be approximated by:

$$\log f(\boldsymbol{x}) \approx \frac{1}{2}\boldsymbol{x}^T H \boldsymbol{x} + G^T \boldsymbol{x} + \log f(0), \tag{6.11}$$

where H is the Hessian matrix and G is the gradient vector at $\log f(\mathbf{0})$. If we substitute this approximation into the Hamiltonian equation (Equation (6.8)) using $\frac{\partial \log f(\boldsymbol{x})}{\partial \boldsymbol{x}} \approx H\boldsymbol{x} + G$ and combine the two differential equations, we get:

$$\frac{\partial^2 \boldsymbol{x}(t)}{\partial t^2} = AH\boldsymbol{x}(t) + AG. \tag{6.12}$$

The above equation is a standard second-order differential equation system, and has an analytical solution. We start from the one-dimensional case, then generalize it to higher dimensions. Assuming x is a one-dimensional variable, if we let $\alpha = AH$, $\beta = AG$, an analytical solution is:

$$x(t) = \begin{cases} c_1 \exp\left(\sqrt{\alpha}t\right) + c_2 \exp\left(-\sqrt{\alpha}t\right) - \frac{\beta}{\alpha} & \text{if } \alpha > 0 \\ c_1 \cos\left(\sqrt{-\alpha}t\right) + c_2 \sin\left(\sqrt{-\alpha}t\right) - \frac{\beta}{\alpha} & \text{if } \alpha < 0 \\ c_1 t + c_2 + \frac{\beta t^2}{2} & \text{if } \alpha = 0, \end{cases} \tag{6.13}$$

which can be verified by plugging the solution back into the equation. The constant multipliers c_1, c_2 can be obtained by plugging in the initial condition $x(0) = 0$, $x'(0) = Ap(0)$ (where $p(0)$ is sampled from the Gaussian distribution ϕ defined in Equation (6.9)) into the original Hamiltonian equation (Equation (6.8)). Specifically, the constants are:

$$c_1 = \begin{cases} \frac{1}{2}\left(\frac{\beta}{\alpha} + \frac{\hat{p}(0)}{\sqrt{\alpha}}\right) & \text{if } \alpha > 0 \\ \frac{\beta}{\alpha} & \text{if } \alpha < 0 \\ \hat{p}(0) & \text{if } \alpha = 0 \end{cases}, c_2 = \begin{cases} \frac{1}{2}\left(\frac{\beta}{\alpha} - \frac{\hat{p}(0)}{\sqrt{\alpha}}\right) & \text{if } \alpha > 0 \\ \frac{\hat{p}(0)}{\sqrt{-\alpha}} & \text{if } \alpha < 0 \\ 0 & \text{if } \alpha = 0, \end{cases} \tag{6.14}$$

where we denote $\hat{p}(0) = Ap(0)$ for clarity.

To illustrate, since the inverse mass A is required to be positive, if the second derivative

H is strictly negative, we consider the $\alpha < 0$ case and the trajectory $x(t)$ becomes a linear combination of a cosine curve and a sine curve, which oscillates in the ridges of the flipped function. On the other hand, if the second derivative is strictly positive, then the trajectory climbs straight up the hill and goes to infinity as t increases.

If x is an N-dimensional vector instead, the general solution of this differential equation system becomes a linear combination of the eigenvectors $\boldsymbol{e}_i$ of the matrix AH:

$$\boldsymbol{x}(t) = \sum_{i=1}^{N} x_i(t)\, \boldsymbol{e}_i, \tag{6.15}$$

where the coefficient $x_i(t)$ is similar to the one-dimensional case (Equation (6.13)), but with α substituted with matrix AH's i-th eigenvalue λ_i, and β and $\hat{p}(0)$ substituted with the projection of the vector AG and $\hat{\boldsymbol{p}}(0)$ on the i-th eigenvector $\boldsymbol{e}_i$, respectively. Again, we can obtain the constant multipliers as in the one-dimensional case by plugging in the initial conditions.

A Gaussian equivalent to the approximation We have derived an analytical trajectory for a fixed initial momentum. However, having the analytical trajectory is not enough. Recall that Hamiltonian Monte Carlo starts by generating a Gaussian distributed momentum $\boldsymbol{p}(0) \sim \mathcal{N}(0, A^{-1})$, and generates a new position proposal $x(T)$ at a fixed time T. Unfortunately, a direct application of the analytical solution to Hamiltonian Monte Carlo using the original acceptance rule (Equation (6.10)) is infeasible. The gradient and Hessian generally would be different at the proposal position, and the time-reversibility would be violated.

An observation from the analytical solution (Equations (6.13) and (6.14)) reveals that the Hamiltonian dynamics are actually linear mappings from the Gaussian distributed variable $\boldsymbol{p}(0)$ to the new position $\boldsymbol{x}(T)$ if we have $t = T$ fixed. This means that $\boldsymbol{x}(T)$ is also Gaussian distributed since Gaussian variables are closed under linear transformations. Therefore, we can generate $\boldsymbol{x}(T)$ using *a single Gaussian distribution*. Furthermore, the probability density of the Gaussian can be used as the transition probability T to compute the Metropolis-Hastings acceptance probability (Equation (6.3)).

Now we will show why the mapping is linear and how to derive the covariance and the mean of $\boldsymbol{x}(T)$. Again we start from the one-dimensional case. If we plug the multipliers c_1 and c_2 (Equation (6.14)) into the analytical solution (Equation (6.13)) and rearrange the

terms in one-dimensional $x(T)$, we have:

$$\begin{aligned} x(T) &= \begin{cases} \left(\frac{\exp\left(\sqrt{\alpha}T\right)-\exp\left(-\sqrt{\alpha}T\right)}{2\sqrt{\alpha}}\right)\hat{p}(0) & \\ \quad +\frac{\beta}{2\alpha}\left(\exp\left(\sqrt{\alpha}T\right)+\exp\left(-\sqrt{\alpha}T\right)-1\right) & \text{if } \alpha > 0 \\ \frac{1}{\sqrt{-\alpha}}\sin\left(\sqrt{-\alpha}T\right)\hat{p}(0) & \\ \quad +\frac{\beta}{\alpha}\left(\cos\left(\sqrt{-\alpha}T\right)-1\right) & \text{if } \alpha < 0 \\ T\hat{p}(0)+\frac{\beta T^2}{2} & \text{if } \alpha = 0 \end{cases} \\ &= s\hat{p}(0)+o \\ &= sAp(0)+o, \end{aligned} \tag{6.16}$$

which is a linear function of $p(0)$ and we denote the scaling coefficient as s and the offset coefficient as o.

For the N-dimensional case, since $\boldsymbol{x}(T)$ is a linear combination of $x_i(T)$ (Equation (6.15)), it is still a linear transform. Moreover, if we write $x_i(T) = s_i \cdot \hat{p}_i(0) + o_i$, where $\hat{p}_i(0)$ is the projection of $\hat{\boldsymbol{p}}(0)$ on the i-th eigenvector $\boldsymbol{e}_i$ of the matrix AH, we can write out the linear transformation in matrix form:

$$\boldsymbol{x}(T) = SA\boldsymbol{p}(0) + \boldsymbol{o}, \tag{6.17}$$

where the matrix S and the vector $\boldsymbol{o}$ can be obtained from the eigenvectors $\boldsymbol{e}_i$, and the coefficients s_i and o_i:

$$S = \sum_{i=1}^{N} s_i \boldsymbol{e}_i, \quad \boldsymbol{o} = \sum_{i=1}^{N} o_i \boldsymbol{e}_i. \tag{6.18}$$

Recall that $\boldsymbol{p}(0)$ is a zero-mean Gaussian variable with covariance A^{-1}. Therefore the covariance matrix Σ and the mean $\boldsymbol{\mu}$ of the Gaussian random variable $\boldsymbol{x}(T)$ are

$$\Sigma = (SA)A^{-1}(SA)^T = SAS^T, \quad \boldsymbol{\mu} = \boldsymbol{o}. \tag{6.19}$$

Multiplying with prior Gaussian In practice, our second order approximation (Equation (6.11)) can be inaccurate when the proposal is far from the current state, or if there are discontinuities such as visibility changes. To compensate for this, we introduce a prior Gaussian distribution with zero mean and isotropic variance using a user specified constant σ^2, and multiply the PDF of it with the PDF of the Gaussian random variable $\boldsymbol{x}(T)$, to effectively place a limit on the maximum variance (which corresponds to the movement of the path in path space).

Another way to think about the prior is that it acts as a regularization term that penalizes

(a) $\sigma^2 = 0.028$
accept rate 28.96%

(b) $\sigma^2 = 0.007$
accept rate 54.02%

(c) $\sigma^2 = 0.001$
accept rate 82.11%

Figure 6-5: We show the effect of the prior Gaussian parameter σ^2 using an inset from the BATHROOM scene (Figure 6-8). (a) High σ^2 results in low acceptance rate and a noisy image. (c) Low σ^2 results in high acceptance rate, but produces correlated noise. (b) We choose a σ^2 so that the acceptance rate falls in the ranges from 50% – 70%.

high variance. If σ^2 is high, then the change of the light path would be large, and the acceptance rate would be lower. On the other hand, if σ^2 is low, then the change of the light path is small, and the acceptance rate would be higher. We show the effects of different σ^2 in Figure 6-5. In our current implementation we manually set σ^2 to achieve a certain acceptance rate (50% to 70%), but it may be possible to automatically adjust the parameter using adaptive MCMC [8]. The final mean $\boldsymbol{\mu}^*$ and covariance Σ^* are

$$\Sigma^* = \left(\Sigma^{-1} + \frac{1}{\sigma^2}\right)^{-1}, \quad \boldsymbol{\mu}^* = \Sigma^* \Sigma^{-1} \boldsymbol{o}. \tag{6.20}$$

Computing acceptance probability In order to apply the Metropolis-Hastings rule (Equation (6.2)) given a current position $\boldsymbol{x}$, we generate a new proposal position $\boldsymbol{y}$ from a Gaussian variable with mean $\boldsymbol{\mu}_x^*$ and covariance matrix Σ_x^* computed using Equation (6.20). Then we compute the mean $\boldsymbol{\mu}_y^*$ and covariance matrix Σ_y^* at the proposal position. The acceptance probability (Equation (6.3)) is computed using the density of Gaussians:

$$\begin{aligned} a\left(\boldsymbol{x} \to \boldsymbol{y}\right) &= \min\left(1, \frac{f\left(\boldsymbol{y}\right) Q\left(\boldsymbol{y} \to \boldsymbol{x}\right)}{f\left(\boldsymbol{x}\right) Q\left(\boldsymbol{x} \to \boldsymbol{y}\right)}\right) \\ &= \min\left(1, \frac{f\left(\boldsymbol{y}\right) \Phi_{\boldsymbol{y}}\left(\boldsymbol{x} - \boldsymbol{y}\right)}{f\left(\boldsymbol{x}\right) \Phi_{\boldsymbol{x}}\left(\boldsymbol{y} - \boldsymbol{x}\right)}\right), \end{aligned} \tag{6.21}$$

where $\Phi_{\boldsymbol{x}}(\boldsymbol{y}-\boldsymbol{x})$ is the Gaussian PDF with covariance Σ_x^* and mean $\boldsymbol{\mu}_x^*$ computed at $\boldsymbol{x}$ (Equation (6.20)). Specifically, if we define $\boldsymbol{z}=\boldsymbol{y}-\boldsymbol{x}$, it is:

$$\Phi_{\boldsymbol{x}}(\boldsymbol{z}) = (2\pi)^{-\frac{N}{2}} |\Sigma_x^*|^{\frac{1}{2}} \exp\left(-\frac{1}{2}(\boldsymbol{z}-\boldsymbol{\mu}_x^*)^T \Sigma_x^{*-1}(\boldsymbol{z}-\boldsymbol{\mu}_x^*)\right). \tag{6.22}$$

$\Phi_{\boldsymbol{y}}(-\boldsymbol{z})$ is defined similarly with covariance and mean computed at $\boldsymbol{y}$.

Setting parameters A and T A remaining question is how to choose the inverse mass matrix A and simulation time T. Previous work in Hamiltonian Monte Carlo suggests setting A to the covariance of the target function [155, 60]. As an example, consider a target function $f(\mathbf{x})$ that is a Gaussian distribution with covariance Σ_f. If we ignore multiplication with the prior, setting A to the covariance of the target function, and setting $T=\pi/2$ will result in a Gaussian mutation (Equation (6.19)) that precisely matches the target function. [2]

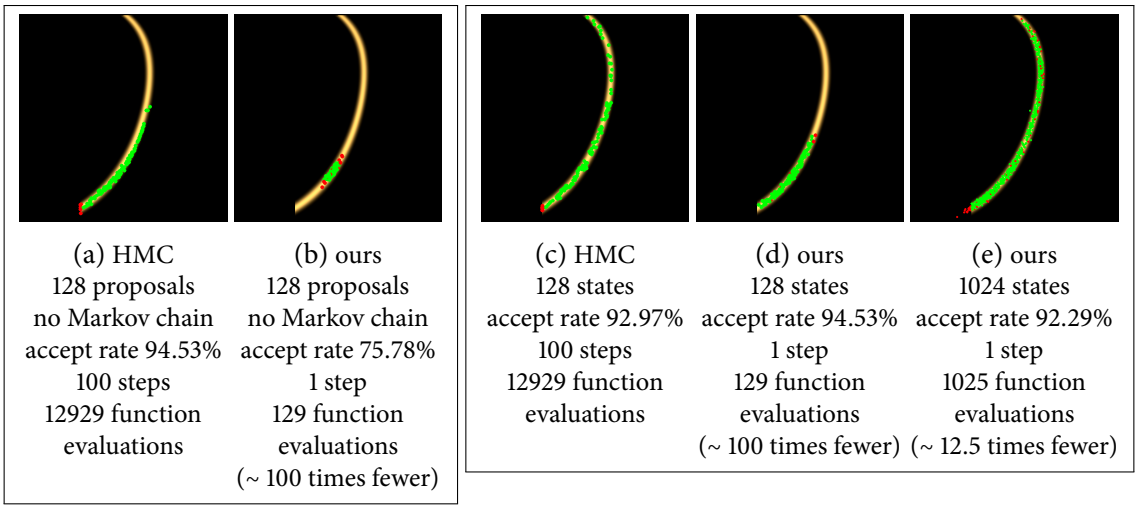
In general, the target function need not be a Gaussian and a global covariance Σ_f may not be sufficient to describe the function. We approximate the covariance locally using the fact that we have the Hessian H of the log of the function. If the target function is a Gaussian, the negative inverse of the Hessian $-H^{-1}$ would exactly be the covariance of the target function. It would be tempting to directly set A to $-H^{-1}$, but the covariance matrix of a Gaussian distribution is required to be positive semidefinite (all eigenvalues need to be positive), and $-H^{-1}$ is not necessarily positive definite in general. We approximate the local covariance of the function by substituting the eigenvalues in $-H^{-1}$ by their absolute values, and set A to the approximated local covariance:

$$A = \sum_{i=1}^{N} \begin{cases} \frac{1}{|\lambda_i^H|}\boldsymbol{e}_i^H & \text{if } \lambda_i^H \neq 0 \\ 0 & \text{otherwise,} \end{cases} \tag{6.23}$$

where $\boldsymbol{e}_i^H$ and λ_i^H are the i-th eigenvector and eigenvalue of H. Finally, we set T to $\frac{\pi}{2}$, as in the Gaussian example above.

The construction of A and T also simplifies the implementation, since A and H share the same set of eigenvectors and A's eigenvalues are the inverse of the absolute value of H's eigenvalues or zero. The eigenvalues λ_i of matrix AH would then be either −1, 1, or 0, depending on the sign of the eigenvalue of H. The magnitudes of the eigenvalues in

[2] In this case, the Hessian H for $\log f$ will simply be $-\Sigma_f^{-1}$, and $\alpha = AH$ will be a negative identity matrix. Therefore, we consider the $\alpha < 0$ case in Equation (6.16), where $s=1$ for $T=\pi/2$, and S is the identity matrix. Therefore, the covariance matrix Σ from Equation (6.19) is given simply by A, leading to a Gaussian distribution with covariance Σ_f, which is exactly the target function. This justifies setting A to the covariance of the target function, and setting $T=\pi/2$.



(a) HMC
128 proposals
no Markov chain
accept rate 94.53%
100 steps
12929 function evaluations

(b) ours
128 proposals
no Markov chain
accept rate 75.78%
1 step
129 function evaluations
(~ 100 times fewer)

(c) HMC
128 states
accept rate 92.97%
100 steps
12929 function evaluations

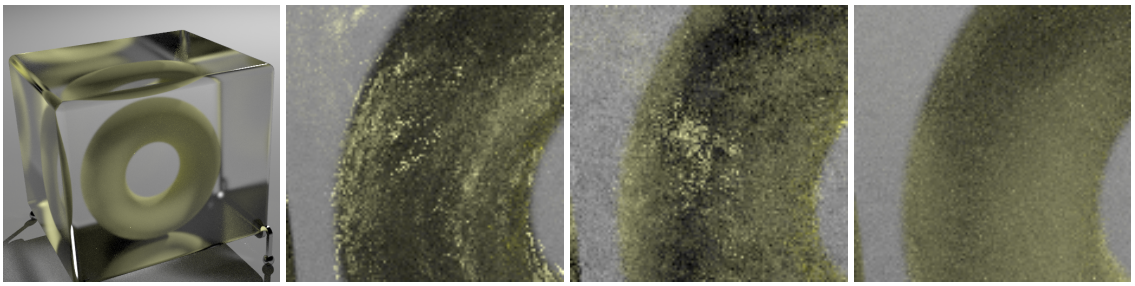
(d) ours
128 states
accept rate 94.53%
1 step
129 function evaluations
(~ 100 times fewer)

(e) ours
1024 states
accept rate 92.29%
1 step
1025 function evaluations
(~ 12.5 times fewer)

Figure 6-6: We compare the sample distribution of the original Hamiltonian Monte Carlo (HMC) method and our method using the zoomed out slices from Figures 6-2 (g) and (h). The left box shows the "proposals" drawn from the current sample position, without running the Markov chain, and the right box shows the actual Markov chain states. While the original HMC is able to generate proposals with high acceptance probability, and over longer trajectories (compare (a) to (b) and (c) to (d)), each proposal in the original HMC requires many steps to compute (100 steps in this case), and each step involves costly ray tracing, shading, and derivative computation. Our method achieves much better space coverage using a single step (as opposed to the 100 steps in the original HMC), and requires an order of magnitude fewer function evaluations (e).

the Hessian H (and hence A) are still taken into consideration when sampling from the momentum using the inverse mass matrix A. We show the pseudo-code of our algorithm in Appendix 6.A, which outputs the final mean $\boldsymbol{\mu}^*$ and covariance Σ^* given the gradient G, Hessian H, and prior σ^2.

Finally, we compare the proposals and the Markov chain of original Hamiltonian Monte Carlo with a leapfrog integrator, and our Hessian-HMC method in Figure 6-6, using a 2D slice in the RING scene (Figure 6-2). We use a step size of 0.0005 with 100 steps for the leapfrog numerical integrator in Hamiltonian Monte Carlo, and we set the prior Gaussian $\sigma^2 = 0.01$ for our method. The target acceptance rate is set higher because the dimensionality of the function is low [155]. Although original Hamiltonian Monte Carlo is able to use longer trajectories to explore the space more thoroughly with the same number of samples, a single sample in Hamiltonian Monte Carlo requires 100 steps of ray tracing, shading and derivatives computation. Choosing a bigger step size or smaller step number for HMC may result in energy loss or inferior space exploration efficiency, and this parameter of original Hamiltonian Monte Carlo is notoriously hard to tune. Our H²MC method can explore the space better using an order of magnitude fewer function evaluations (Figure 6-6(e)).



(a) TORUS (b) Kelemen et al. [111], isotropic proposal (c) Ours, isotropic proposal (d) Ours, H²MC proposal

Figure 6-7: We compare our new parameterization to Kelemen et al.'s parameterization on the TORUS scene, with a diffuse torus inside a glossy glass cube lit by a point light. The left image is computed using 5000 samples per pixel using our method, and the three insets are computed with 256 samples per pixel. The original parameterization incurs correlation between screen space and the outgoing sample directions on the glass and the torus, creating streaks on the torus. Our new parameterization greatly reduces this correlation. Our Hessian-Hamiltonian proposal further improves the sampling efficiency dramatically.

6.4 Implementation

We implement our method in a stand-alone renderer with the Embree ray tracing engine [220]. We implement an embedded automatic differentiation compiler in C++ that overloads all the common functions and operators, and compiles the gradient and Hessian of the path throughput function into an ispc [169] kernel. The renderer supports the Phong BRDF, the microfacet refraction model [222], point and area light sources, environment maps, and linear object motion. Each sample in the Markov chain represents a single light path that connects the light to the camera. As in most previous Markov chain Monte Carlo rendering methods, we employ multiple mutation strategies to better cover different types of light paths. Specifically, we adopt three different types of mutation strategies: a multiplexed *large step* mutation [111, 74], a novel modified *small step* perturbation, and a lens perturbation. The large step mutation is responsible for making large jumps between different disconnected components of light paths, the small step perturbation is responsible for making a small change to all dimensions of the function, and finally the lens perturbation changes only part of the light path to alleviate difficult visibility issues. We apply the H²MC sampling on the small step and the lens perturbation to explore the local structure of the path throughput function. In the rest of this section, we address some technical details of the implementation.

Multiplexed large step mutation To ensure the ergodicity of the Markov chain, that is, to ensure we have a strictly positive probability to sample all light paths with non-zero contribution, we include a large step mutation to generate a proposal light path that is completely

independent of the current sample. Our large step mutation is a hybrid between Multiplexed Metropolis Light Transport (MMLT) [74] and Kelemen et al.'s mutation. Specifically, like MMLT, the state of our Markov chain only represents one of the N^2 pairs connections (where in Kelemen style the state would be the sum of all connections). However, instead of choosing path length and subpath length a priori as in MMLT, we sample all pairs of connections of a bidirectional path tracer, and probabilistically pick one based on their contributions weighted by multiple importance sampling (similar to Multiple-try Metropolis [136] and importance resampling [185]). Comparing to MMLT, this has the benefit of stratification, since we always sample all path and subpath lengths instead of randomly choosing one. Comparing to Kelemen's mutation, during the other local perturbations, we only keep a single subpath in a bidirectional path tracer, therefore we benefit from multiple importance sampling just like MMLT. We notice that the acceptance rate of large steps significantly increases in difficult scenes compared to MMLT when using our approach.

H² small step perturbation We adopt a modified version of the *small step* perturbation [111] as the main component to explore the path throughput function locally. In Kelemen et al.'s work, the light paths are represented as the random numbers that are used to generate them. The perturbation is done by making small changes to the random numbers, and results in a new light path. We make two modifications to the parameterization.

First, we classify the surfaces into specular and non-specular by applying a user-defined threshold on the roughness. If the surface is near-specular, the outgoing directions are parameterized using the random numbers. On the other hand, if the surface is non-specular, the outgoing directions are parameterized using the global directions expressed in absolute spherical coordinates. We found that this change improves sampling efficiency because the correlation between the dimensions is reduced. Kelemen's parameterization handles specular surfaces well, because importance sampling captures the peak of the target function well. On the other hand, the local parameterization introduces extra correlation between dimensions, because the outgoing direction depends on the normal of the surface, and the normal depends on the previous outgoing direction. The parameterization change is beneficial because H²MC is invariant to linear parametrization changes, while the parameterization change is non-linear. We show a comparison of the original parameterization with the new one in Figure 6-7.

Second, if the light path hits a light source without next event estimation (that is, no explicit connection is made), we substitute the parameterization of the last outgoing direction, to the position on the light source, so that the perturbation is more likely to hit the light source (a limited form of Reversible Jump Markov Chain Monte Carlo [162, 160, 23]). We assume a pinhole camera in our implementation, but the second change can also apply in

	H^2MC	MMLT	MEMLT	HSLT
Bathroom	610	1288	600	331
Kitchen	5169	12453	4749	3319
Balls	2943	8554	2961	N/A
Cars	1576	5361	1422	N/A

Table 6.1: Sample count per pixel of each method for the equal-time comparisons.

the case when the light path starts from the light source and hits the camera lens without explicit connection. The new parameterization represents the sample position $\boldsymbol{x}$ in the H²MC sampling.

The time dimension is treated the same as other dimensions. The generality of H²MC sampling makes it agnostic to the underlying representation. This enables us to detect the correlation between time and other dimensions, which was not considered in previous Markov chain Monte Carlo rendering methods.

H² lens perturbation Consider light paths involving small and flat surfaces. If we mutate the whole path, chances are high that we will miss the surfaces and result in zero contribution. A better strategy for these light paths is to perturb only a subset of the full path, and keep the rest of the vertices fixed. We implement the lens perturbation in the original Metropolis Light Transport algorithm [216], which mutates only the lens subpath. For lens perturbation, the sample position $\boldsymbol{x}$ in the H²MC sampling is the two dimensional image coordinate.

6.5 Results and Discussion

We compare against three other MCMC rendering methods: Multiplexed MLT (MMLT) [74], Manifold Exploration MLT (MEMLT) [97], and the improved Half-vector Space Light Transport (HSLT) [75]. MMLT is a general rendering algorithm that does not assume any particular lighting effect, but its isotropic mutation makes it inefficient on difficult light paths such as highly-glossy transports. We compare to MMLT to show the efficiency of the anisotropic proposal sampling. MEMLT and HSLT are two rendering algorithms dedicated to specular and glossy transport by using first-order derivatives of the half-vectors. They can efficiently resolve difficult specular light paths, but often produce noisy results on highly-curved surfaces. Furthermore, since they assume a specific lighting scenario, they cannot resolve difficult moving caustics, and usually result in ghosting artifacts (Figures 6-1 and 6-11). We did not compare to HSLT on the scenes with motion blur because their implementation does not allow it. We render four scenes – Bathroom (1280 × 720), Kitchen (1024 × 576), Balls

(768 × 576), Cars (768 × 576) – with different lighting, material, and geometry configurations (Figure 6-1 and Figures 6-8 to 6-11).

For MMLT we use our own implementation, for MEMLT and HSLT we use the implementation in the Mitsuba [95] renderer. HSLT is used with the lens perturbation because in our experiments it results in better images. The comparisons are equal-time using an Intel Core i7-4770 at 3.40GHz using 4 cores. The maximum path length is set to 7. References are rendered using the PSSMLT [111] implementation in Mitsuba and rendered for 2-3 days on a 64 core machine, except that the reference for the Cars is rendered using our method for roughly 15 hours on the 4 core machine (PSSMLT did not converge in 2-3 days computation). We show the sample count per pixel of each method in each scene in Table 6.1. In general our method is 2-3.5 times slower per sample than MMLT because of the derivatives and Gaussian computation, and is about the same speed as MEMLT. HSLT is slower than MEMLT because it works on a higher-dimensional manifold.

Bathroom Figure 6-8 shows an equal-time (10 minutes) comparison on the *bathroom* scene with multiple glossy-to-glossy transports lit by a distant area light. For this particular scene, only indirect illumination is shown to highlight the differences between the algorithms. MMLT generates noisy results because of their isotropic mutation distribution. MEMLT and HSLT do generally well, but produce noisy results on high curvature surfaces because they use a first-order approximation on the surface. Our method is able to capture the local structure of the function and generates accurate results.

To demonstrate the anisotropic proposal distribution of our method, we visualize the screen space slice of the contribution of some light paths and the slice of our Gaussian approximation in Figure 6-9. Our method is able to adapt to the sparse and sharp path contribution function, and fall back to isotropic sampling when the contribution function is smooth. MEMLT and HSLT often fail to capture small screen space features, because they isotropically sample some dimensions first, and such sampling often misses the feature. Note that our method adapts to all dimensions, and we only show the screen space slice for the sake of visualization.

Kitchen Figure 6-10 shows an equal-time (1 hour) comparison on the *kitchen* scene with complex materials and a difficult geometry configuration lit by four area lights close to the table. This is a challenging scene and the reference rendered by PSSMLT is still slightly noisy after 2 days of computation on a 64 core machine. MMLT produces spiky noise because some glossy-to-glossy light paths have small and high-contribution regions. MEMLT and HSLT generate noisy results on small and highly curved surfaces. Our method is able to follow the

small image features closely, producing smoother results.

In general, light paths involving highly curved surfaces can be troublesome for MEMLT and HSLT, which only use first derivatives. Both of them need to start from an initial subpath, then iteratively converge to the new light path on the manifold. The light paths involving curved surfaces often have narrow contribution areas, and are highly non-linear. It is likely that the initial subpath will miss the highlight entirely, making it impossible to converge to a new light path. Even if the initial subpath hits the highlight, it could take many iterations to converge due to the non-linearity. In contrast, the second derivatives along with the anisotropic Gaussian mutation enable us to generate the proposal path directly with respect to the local shape of the function, avoiding the convergence issue.

Balls Figure 6-11 shows a 30 minute rendering of the *balls* scene, which consists of three moving near-specular glass balls lit by a point light. MMLT is unable to resolve the difficult specular-diffuse-specular paths inside the moving balls and the caustics on the table. While MEMLT excels at resolving the specular light paths given the time fixed, it relies on seeding to sample the time dimension, which causes the ghosting artifacts on the balls. Our method is able to capture correlation between the time and the path-space, so that it can efficiently sample the difficult moving caustics and specular highlights.

Cars. Figure 6-1 shows a 20 minute rendering of the *cars* scene, with a static car and a moving car lit by an area light. This is a challenging scene because of the hard-to-find specular-diffuse-specular (SDS) light paths between the car interior and the near-specular window. MMLT has a hard time finding the specular light paths, and is often trapped in local modes, producing streaks on the image. MEMLT is able to resolve the static SDS paths more efficiently, but produces ghosting artifacts since it does not move in the time dimension. Our method moves in all dimensions and generates smooth results.

6.5.1 Limitations and Future Work

Integrating our method into an existing renderer requires some work, because we need to automatically differentiate the shaders. However, once automatic differentiation has been set up, it is easier to integrate other distributed effects such as motion blur. Automatic differentiation could also be helpful for the shaders/integrators that require the derivatives of the light path (e.g. ray differentials). The production renderer Arnold [118] uses forward-mode automatic differentiation to compute ray differentials.

As with most Markov-chain Monte Carlo rendering algorithms, high frequency visibility changes can significantly lower the efficiency. Our Gaussian prior reduces this effect but tiny

geometry can still cause problems. In addition to visibility changes, there can also be some pathological cases where the path contribution function is extremely noisy. For example, multiple-bounce reflections involving glossy surfaces with high frequency displacement maps. In these cases the derivatives become unreliable, and our method might start to produce correlated noise or have low acceptance rate. Combining our method with the visibility gradient introduced in Chapter 5, or the recent cone fitting approach [159] could be interesting future work. Proper prefiltering of geometry and texture is also important for the derivatives to be well-behaved (e.g. [138]).

We also observe that light transport integration involves both global and local exploration challenges. We need to globally find high-contribution regions, and then locally sample them despite their narrowness. Our method improves local sampling, but it still needs seed paths that are globally reasonably well distributed. Combining recent data-driven methods, e.g., [219, 175], with local perturbation is one possible research direction.

Finally, since the derivatives and covariance computation incurs extra overhead, for relatively simple scenes and BSDFs where ray casting is cheap and isotropic mutation is sufficient, the adaptiveness of our method may not be worth the cost.

6.6 Conclusion

We presented a novel Hessian-based Hamiltonian Monte Carlo method and applied it to light transport simulation. By introducing Hamiltonian dynamics, we are able to sample from the local quadratic representation that does not define a distribution. Our method can capture the local correlation of the path throughput function, making it suitable for rendering difficult lighting scenarios such as the combination of glossy-to-glossy transport and motion blur. We anticipate that the method's generality will make it possible to render a wider variety of effects such as retroreflective materials, spectral effects, and participating media.

6.A Pseudo-code for H²MC

Given the gradient G and the Hessian H of the log target function $\log f(x)$, and a user parameter σ^2, our method outputs an anisotropic Gaussian distribution $\Sigma^*, \boldsymbol{\mu}^*$. The following page shows the pseudo code of this procedure. Note that we simplify the algorithm using the fact that the inverse mass matrix A and H have the same set of eigenvectors.

procedure H2MC(G, H, σ^2) ▷ gradient, Hessian, and prior
  $N \leftarrow$ dimension of the target function
  $T = \frac{\pi}{2}$ ▷ Simulation time
  for $i \leftarrow 1, N$ **do** ▷ Eigendecomposition of H
    $\boldsymbol{e}_i^H \leftarrow i$-th eigenvector of H
    $\lambda_i^H \leftarrow i$-th eigenvalue of H
  end for
  $A = 0_{N\times N}$ ▷ Initialize with zero matrix
  for $i \leftarrow 1, N$ **do** ▷ Construction of A
    $\boldsymbol{e}_i^A \leftarrow \boldsymbol{e}_i^H$
    if $|\lambda_i^H| > \epsilon$ **then** ▷ ϵ is set to a small number.
      $\lambda_i^A \leftarrow \frac{1}{|\lambda_i^H|}$
    else
      $\lambda_i^A \leftarrow 0$
    end if
    $A = A + \lambda_i^A \boldsymbol{e}_i^A$
  end for
  for $i \leftarrow 1, N$ **do** ▷ Eigendecomposition of the matrix AH
    $\boldsymbol{e}_i \leftarrow \boldsymbol{e}_i^H$
    if $|\lambda_i^H| > \epsilon$ **then**
      $\lambda_i \leftarrow \frac{\lambda_i^H}{|\lambda_i^H|}$ ▷ $\lambda_i^A = \frac{1}{|\lambda_i^H|}$
    else
      $\lambda_i \leftarrow 0$
    end if
  end for
  $S = 0_{N\times N}$
  $\boldsymbol{o} = 0_{N\times 1}$
  for $i \leftarrow 1, N$ **do** ▷ Scales and offsets (Equation (6.16))
    $\alpha \leftarrow \lambda_i$ ▷ AH's i-th eigenvalue
    $\beta \leftarrow \lambda_i^A G^T \boldsymbol{e}_i$ ▷ Projection of AG on $\boldsymbol{e}_i$
    if $\lambda_i > 0$ **then**
      $s_i \leftarrow \frac{\exp\left(\sqrt{\alpha}T\right) - \exp\left(-\sqrt{\alpha}T\right)}{2\sqrt{\alpha}}$
      $o_i \leftarrow \frac{\beta}{2\alpha}\left(\exp\left(\sqrt{\alpha}T\right) + \exp\left(-\sqrt{\alpha}T\right) - 1\right)$
    else if $\lambda_i < 0$ **then**
      $s_i \leftarrow \frac{1}{\sqrt{-\alpha}}\sin\left(\sqrt{-\alpha}T\right)$
      $o_i \leftarrow \frac{\beta}{\alpha}\left(\cos\left(\sqrt{-\alpha}T\right) - 1\right)$
    else
      $s_i \leftarrow T$
      $o_i \leftarrow -\frac{\beta T^2}{2}$
    end if
    $S = S + s_i \boldsymbol{e}_i$ ▷ Equation (6.18)
    $\boldsymbol{o} = \boldsymbol{o} + o_i \boldsymbol{e}_i$
  end for
  $\Sigma = SAS^T$ ▷ Equation (6.19)
  $\boldsymbol{\mu} = \boldsymbol{o}$
  $\Sigma^* = \left(\Sigma^{-1} + \frac{1}{\sigma^2}\right)^{-1}$ ▷ Prior multiplication (Equation (6.20))
  $\boldsymbol{\mu}^* = \Sigma'\Sigma\boldsymbol{\mu}$
  return $\Sigma^*, \boldsymbol{\mu}^*$
end procedure

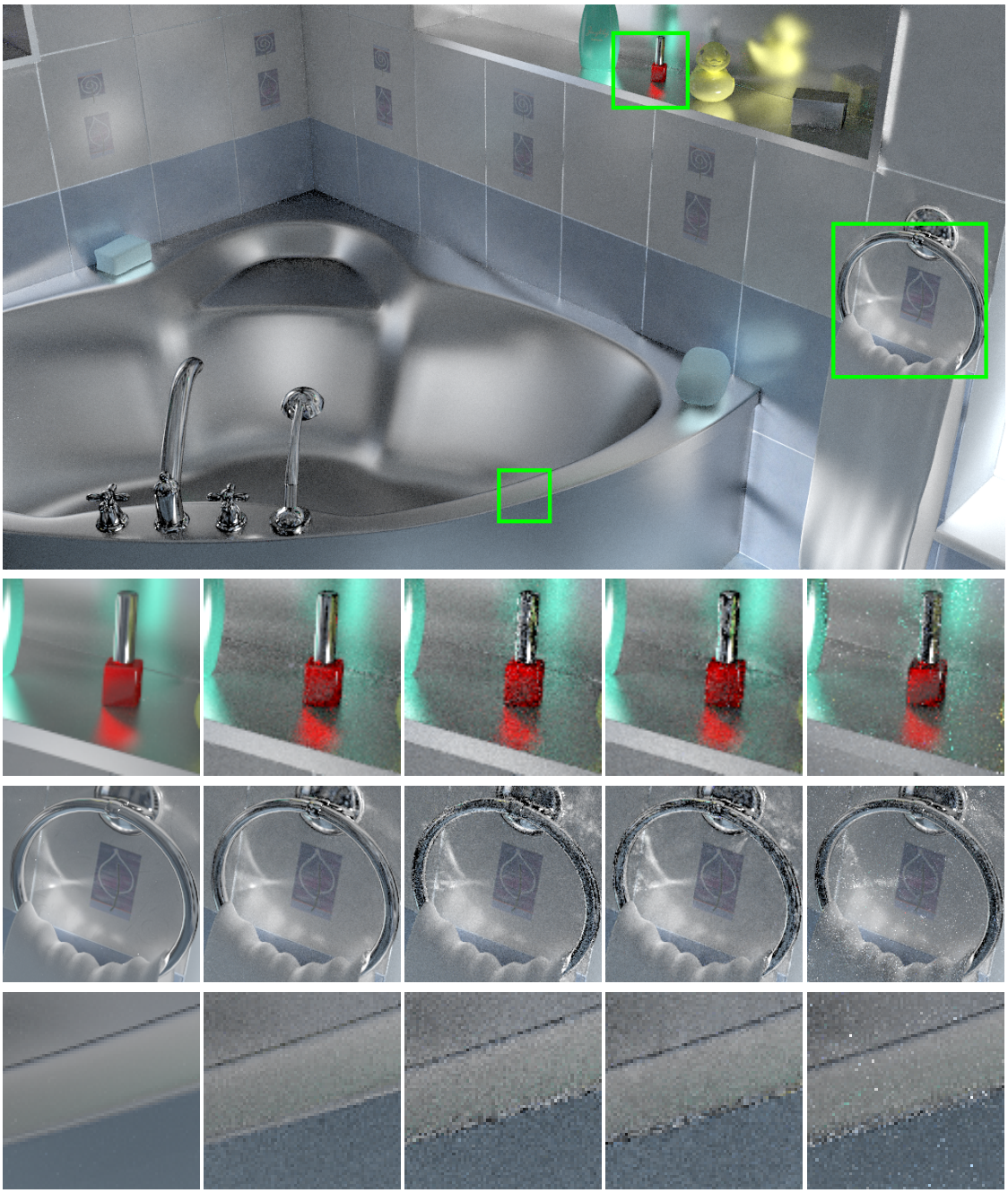


Figure 6-8: Bathroom: An equal-time (10 minutes) comparison on the *bathroom* scene with multiple glossy reflections lit by a distant area light. The top image is generated by our method in 10 minutes. Our method achieves less noisy results on highly curved glossy surfaces and the caustics because we can adapt to the curvatures of the surfaces using second-order derivatives.

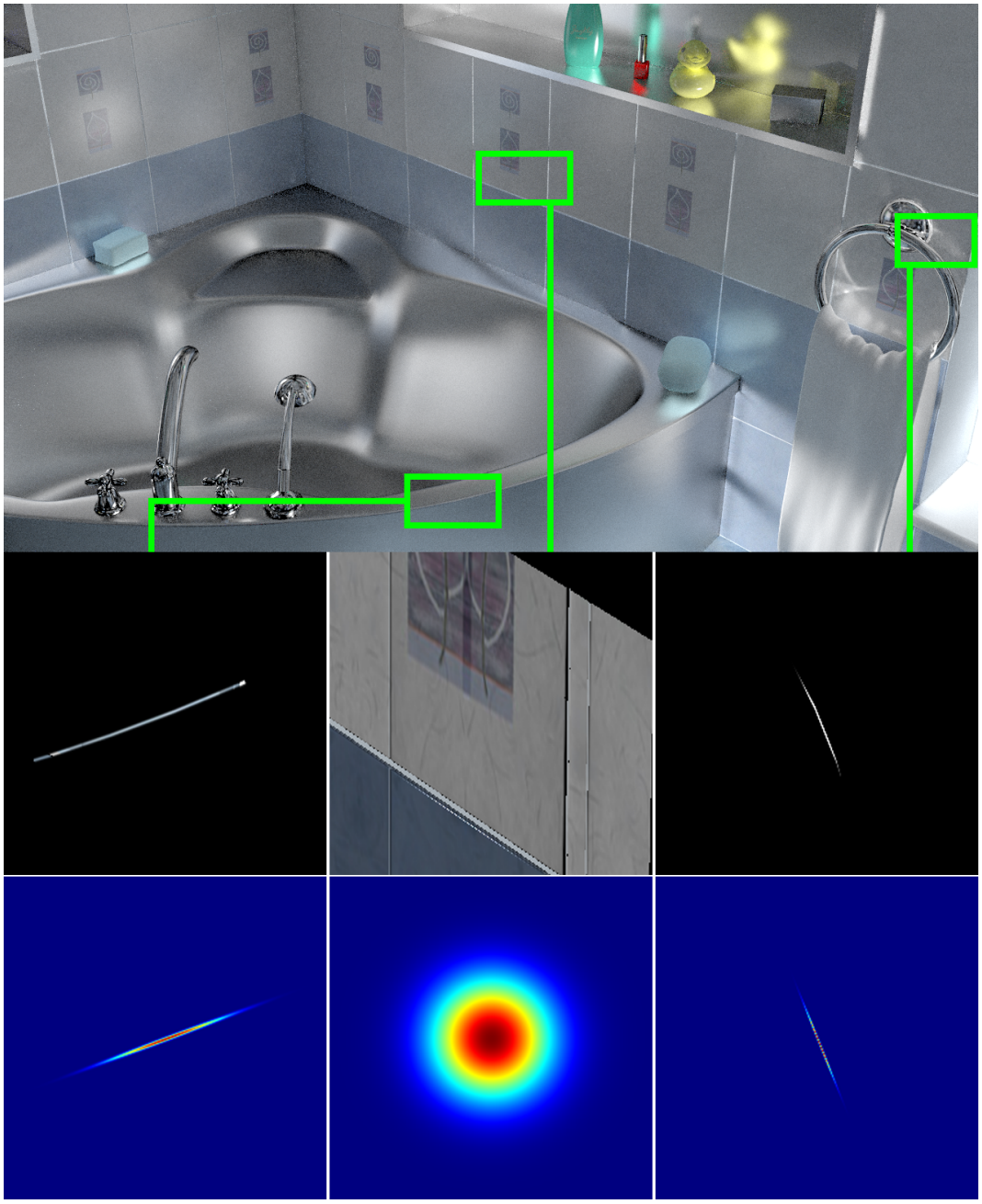


Figure 6-9: We visualize the screen space slice of the contribution of three different light paths and our Gaussian approximation $\mathcal{N}(\boldsymbol{\mu}^*, \Sigma^*)$ in the BATHROOM scene. The center row of the insets shows the contribution of perturbing the light path in the screen space. The left column shows a 4 bounce glossy reflection light path, the center column shows a 3 bounce diffuse reflection light path, and the right column shows a 3 bounce caustic light path caused by the metal towel ring. Glossy/specular transport results in sparse and anisotropic contributions, which are hard to sample using isotropic mutations. The bottom row shows our Gaussian approximation projected onto the screen space. The approximation matches the sharp contribution function and falls back to isotropic sampling when the contribution is smooth. Note that our method is anisotropic in all sampling dimensions, and we only show the screen space slices for visualization purposes.

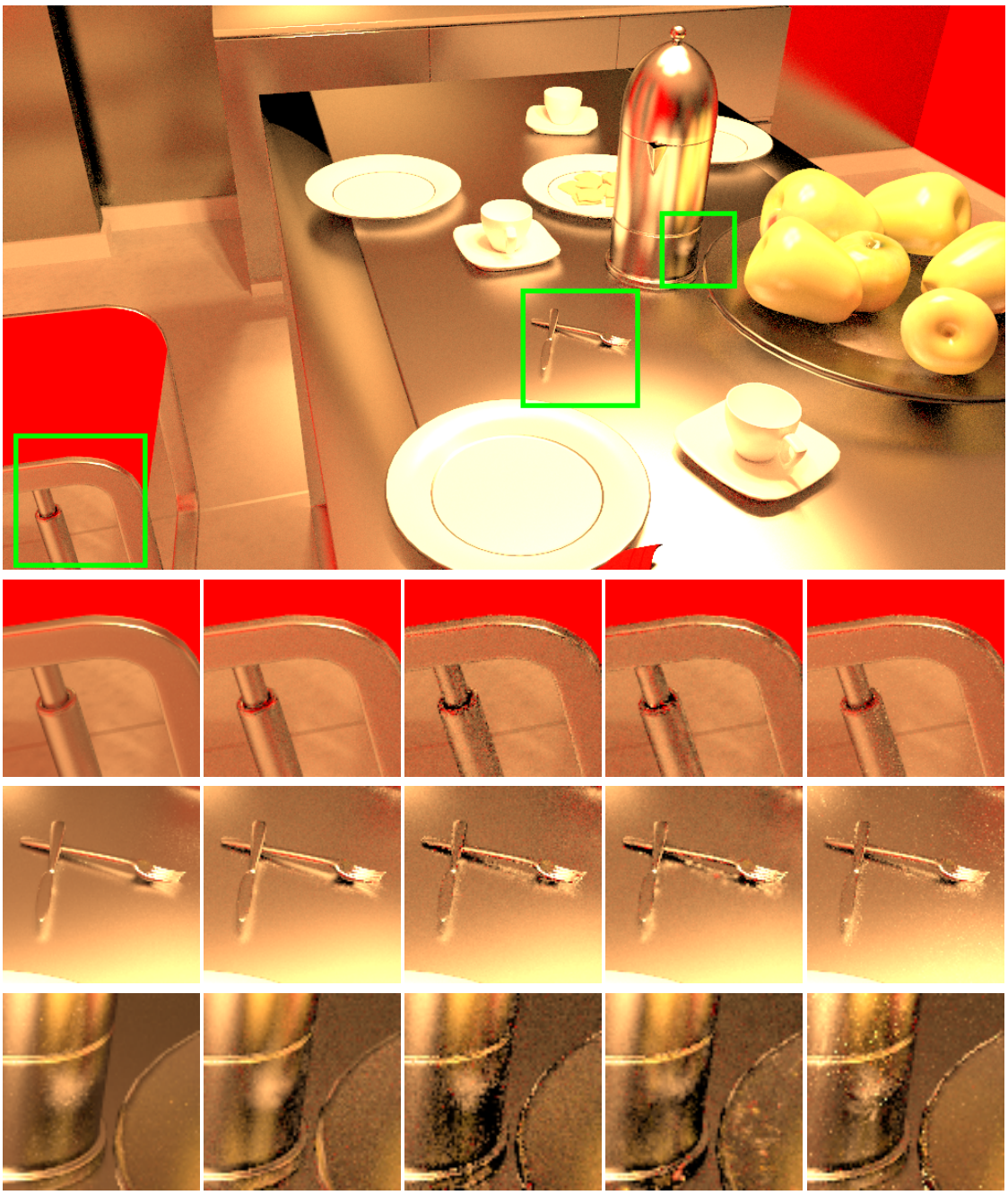


Figure 6-10: KITCHEN: An equal-time (1 hour) comparison on the *kitchen* scene with complex material and geometry configuration lit by four area lights right above the table. The top image is generated by our method in an hour. This is a challenging scene and the reference rendered by PSSMLT is still slightly noisy after 2 days of computation on a 64 core machine. Our method excels at following the small features of the image such as the fork and the knife on the table, or the edges on the chair. It is also good at following the multiple glossy reflections on the highly curved surfaces such as the reflection on the flask.

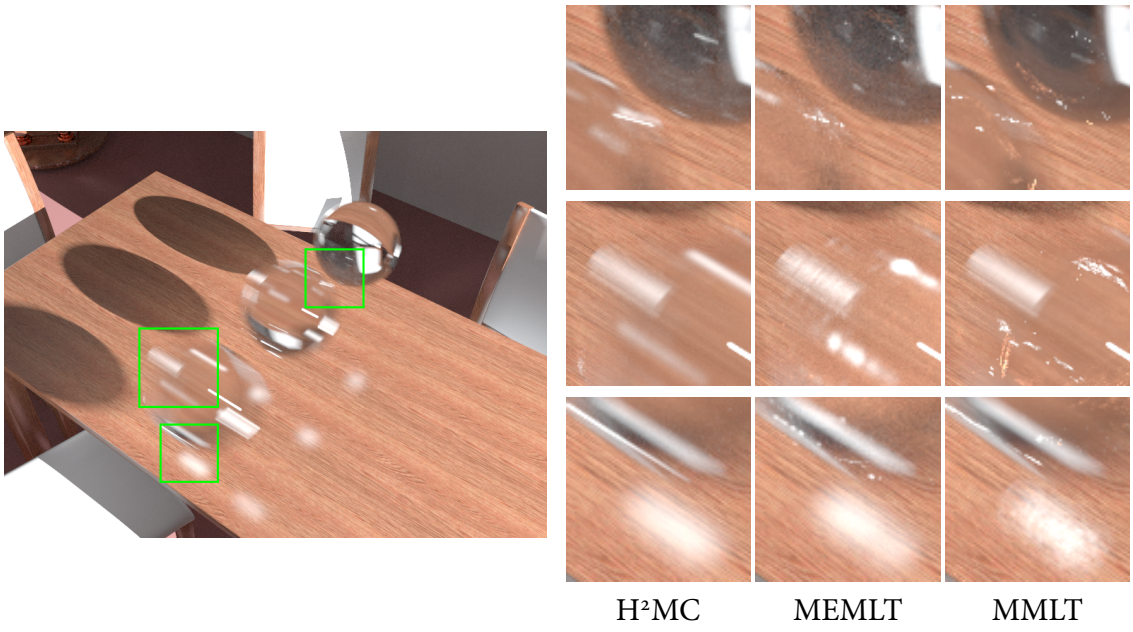


Figure 6-11: BALLS: 30 minute rendering of the *balls* scene, which consists of three moving near-specular glass balls lit by a point light. The left image is generated by our method in 30 minutes. The moving balls show complex patterns with a combination of reflection from the room and the resulting caustics on the table. Neither MMLT nor MEMLT are able to efficiently resolve the moving features within the given time budget. Our method is able to closely follow the specular highlights and caustics in the glass because it detects the correlation between the time domain and path-space.

7 | Conclusion and Future Vision

This dissertation introduced three very different tools related to computing and applying derivatives for computer graphics, image processing, and deep learning applications. Applying derivatives is desirable, but also challenging. At the system level, accounting for parallelism and locality, while preserving expressiveness of a programming language, is not trivial. At the algorithm level, resolving discontinuities, making use of first- or higher-order derivatives often require domain-specific knowledge. The three tools we introduced tackled these challenges in different ways, and are important first steps toward making all programs differentiable.

In the future, my vision is that the distinction between deep learning and traditional algorithms will become even blurrier than they are at the moment. Neural network architectures will become more and more sophisticated, and traditional methods will become more data-driven. For problems where data is limited, it is useful to apply our prior knowledge by formulating a forward model. A key to bridge the gap between deep learning and traditional methods is the generality of automatic differentiation, and the challenges that arose when developing the methods in this dissertation will repeatedly appear when trying to differentiate other programs.

I envision the following future research directions to be important:

Fully differentiable computer graphics While we have been successful at differentiating physically-based rendering (Chapter 5 and 6) and a small fluid simulation example (Chapter 4), these are only subsets of the whole field of computer graphics. It would be desirable to make the whole 3D modeling, simulation, and rendering pipeline differentiable. As shown throughout this dissertation, having fully differentiable pipelines enables data-driven training and inverse inference. As computer graphics models the world with tremendous detail and principled simulation, it would have enormous use in real-world applications.

Automatic differentiation compilers for other computation While we can automatically generate efficient gradient code for image processing and deep learning operators (Chapter 4), the programming model is still relatively limited compared to a fully general compiler. This is

necessary to achieve high performance while writing concise code. However, it is important to cover other computation for automatic differentiation, such as tree traversal, sparse or graph data structures, sorting, etc. As we did in Chapter 4, we need to properly hint the compiler about the computation patterns involved in order to reason about the computation and generate efficient code. I argue that we need more domain specific languages for high performance automatic differentiation.

Generalizing differentiation Derivatives are only one possibility for representing the neighborhood of a point in a function. For example, the Fourier transform and wavelets also characterize the smoothness of a function, with non-local information. While using derivatives are better than treating the program as a black box, it is natural to question whether derivatives are the most useful information we can retrieve from a program. After all, automatic differentiation is merely a program transformation. Finding other transformations of programs that benefit the tasks we are interested in is an interesting direction to pursue. For example, is it possible to find a transformation that is an approximation to the derivatives, but cheaper to compute, or more robust to high frequency changes of the function? Mathematics-wise, generalizations of derivatives have been proposed. Discrete calculus studies functions with integer inputs and real number outputs. Subgradients are commonly used in optimization theory. Systems-wise, program synthesis techniques [128] have been proven to be useful for finding program transformations.

Local minimum, overparametrization, and prefiltering When dealing with function landscapes that are noisy and bumpy, derivative-based methods are known to be unstable. Deep learning does not seem to suffer from this, most likely due to their large amount of parameters (e.g. [53, 110]). Figuring out how to overparametrize traditional algorithms is the key to truly fuse the two domains. Prefiltering in signal processing could also play an important role: if we can smooth out a function *before* we sample it, it will make the derivatives much more well-behaved. Yang and Barnes [232] hinted on how to do this automatically for shader programs in computer graphics, but generalizing their approach to arbitrarily high-dimensional functions requires future research.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] Andrew Adams, Natasha Gelfand, Jennifer Dolson, and Marc Levoy. Gaussian kd-trees for fast high-dimensional filtering. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 28(3):21, 2009.

[3] Miika Aittala, Timo Aila, and Jaakko Lehtinen. Reflectance modeling by neural texture synthesis. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 35(4):65:1–65:13, 2016.

[4] Miika Aittala, Tim Weyrich, and Jaakko Lehtinen. Practical SVBRDF capture in the frequency domain. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 32(4):110:1–110:12, 2013.

[5] Miika Aittala, Tim Weyrich, and Jaakko Lehtinen. Two-shot SVBRDF capture for stationary materials. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 34(4):110:1–110:13, 2015.

[6] Shun-ichi Amari. Neural learning in structured parameter spaces-natural riemannian gradient. In *Advances in Neural Information Processing Systems*, pages 127–133, 1997.

[7] Luke Anderson, Tzu-Mao Li, Jaakko Lehtinen, and Frédo Durand. Aether: An embedded domain specific sampling language for Monte Carlo rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 36(4):99:1–99:16, 2017.

[8] Christophe Andrieu and Johannes Thoms. A tutorial on adaptive MCMC. *Statistics and Computing*, 18(4):343–373, 2008.

[9] James Arvo. The irradiance Jacobian for partially occluded polyhedral sources. In *SIGGRAPH*, pages 343–350. ACM Press/Addison-Wesley Publishing Co., 1994.

[10] Anish Athalye, Logan Engstrom, Andrew Ilyas, and Kevin Kwok. Synthesizing robust adversarial examples. In *International Conference on Machine Learning*, pages 284–293, 2018.

[11] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Code Generation and Optimization*. IEEE, Feburary 2019.

[12] Jonathan T Barron and Jitendra Malik. Shape, illumination, and reflectance from shading. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(8):1670–1687, 2015.

[13] Jonathan T Barron and Ben Poole. The fast bilateral solver. In *European Conference on Computer Vision*, pages 617–632, 2016.

[14] Bruce Guenther Baumgart. Geometric modeling for computer vision. Technical report, Stanford University, 1974.

[15] Laurent Belcour, Cyril Soler, Kartic Subr, Nicolas Holzschuch, and Frédo Durand. 5D covariance tracing for efficient defocus and motion blur. *ACM Trans. Graph.*, 32(3):31:1–31:18, 2013.

[16] Bradley Bell. CppAD: A package for differentiation of C++ algorithms, 2003–2019.

[17] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Python for Scientific Computing Conference (SciPy)*, 2010.

[18] Michael Betancourt. A general metric for Riemannian manifold Hamiltonian Monte Carlo. In *Geometric Science of Information*, pages 327–334, 2013.

[19] Michael Betancourt. A geometric theory of higher-order automatic differentiation. *arXiv:1812.11592*, 2018.

[20] Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies – a comprehensive introduction. *Natural computing*, 1(1):3–52, may 2002.

[21] Shalabh Bhatnagar, HL Prasad, and LA Prashanth. *Stochastic recursive algorithms for optimization: simultaneous perturbation methods*, volume 434. Springer, 2012.

[22] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR: Automatic differentiation in a source translator environment. In *International Symposium on Symbolic and Algebraic Computation*, pages 294–302, 1992.

[23] Benedikt Bitterli, Wenzel Jakob, Jan Novák, and Wojciech Jarosz. Reversible jump Metropolis light transport using inverse mappings. *ACM Trans. Graph.*, 37(1):1:1–1:12, 2017.

[24] Volker Blanz and Thomas Vetter. A morphable model for the synthesis of 3D faces. In *SIGGRAPH*, pages 187–194. ACM Press/Addison-Wesley Publishing Co., 1999.

[25] Adrien Bousseau, Emmanuelle Chapoulie, Ravi Ramamoorthi, and Maneesh Agrawala. Optimizing environment maps for material depiction. *Comput. Graph. Forum (Proc. EGSR)*, 30(4):1171–1180, 2011.

[26] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[27] Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. CRC press, 2011.

[28] Antoni Buades, Bartomeu Coll, and J-M Morel. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition*, volume 2, pages 60–65. IEEE, 2005.

[29] Brent Burley. Physically-based shading at Disney. In *SIGGRAPH Course Notes. Practical physically-based shading in film and game production.*, volume 2012, pages 1–7. ACM, 2012.

[30] Vladimir Bychkovsky, Sylvain Paris, Eric Chan, and Frédo Durand. Learning photographic global tonal adjustment with a database of input / output image pairs. In *Computer Vision and Pattern Recognition*. IEEE, 2011.

[31] Pierre BÃľnard and Aaron Hertzmann. Line drawings from 3D models. *arXiv:1810.01175*, 2018.

[32] Bob Carpenter. Typical sets and the curse of dimensionality. `https://mc-stan.org/users/documentation/case-studies/curse-dims.html`. Accessed: 2019-01-20.

[33] Augustin Cauchy. Méthode générale pour la résolution des systemes dâĂŹéquations simultanées. *Comp. Rend. Sci. Paris*, 25(1847):536–538, 1847.

[34] M. T. Chao. A general purpose unequal probability sampling plan. *Biometrika*, 69:653–656, 1982.

[35] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 26(3), 2007.

[36] Jie Chen and Ronny Luss. Stochastic gradient descent with biased but consistent gradient estimators. *arXiv:1807.11880*, 2018.

[37] Min Chen and James Arvo. Theory and application of specular path perturbation. *ACM Trans. Graph.*, 19(4):246–278, 2000.

[38] Tianqi Chen, Emily Fox, and Carlos Guestrin. Stochastic gradient Hamiltonian Monte Carlo. In *International Conference on Machine Learning*, pages 1683–1691, 2014.

[39] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cuDNN: Efficient primitives for deep learning. *arXiv:1410.0759*, 2014.

[40] James H Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[41] Clifford. Preliminary sketch of biquaternions. *Proceedings of the London Mathematical Society*, s1-4(1):381–395, nov 1871.

[42] David Cline, Justin Talbot, and Parris Egbert. Energy redistribution path tracing. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 24(3):1186–1195, 2005.

[43] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, 1982.

[44] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The triangle processor and normal vector shader: A VLSI system for high performance graphics. *Comput. Graph. (Proc. SIGGRAPH)*, 22(4):21–30, jun 1988.

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition*. IEEE, 2009.

[46] Zachary Devito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. Opt: A domain specific language for non-linear least squares optimization in graphics and imaging. *ACM Trans. Graph.*, 36(5):171:1–171:27, 2017.

[47] Timothy Dozat. Incorporating Nesterov momentum into Adam. In *International Conference on Learning Representations – Workshop track*, 2016.

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

[49] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[50] Alejandro Conty Estevez and Christopher Kulla. Importance sampling of many lights with adaptive tree splitting. *ACM Comput. Graph. Interact. Tech. (Proc. HPG)*, 1(2):25:1–25:17, 2018.

[51] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.

[52] Horacio E. Fortunato and Manuel M. Oliveira. Fast high-quality non-blind deconvolution using sparse adaptive priors. *Visual Comput.*, 30(6-8):661–671, 2014.

[53] Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019.

[54] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Computer Vision and Pattern Recognition*, pages 2414–2423. IEEE, 2016.

[55] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle pointsâĂŤonline stochastic gradient for tensor decomposition. In *International Conference on Learning Theory*, pages 797–842, 2015.

[56] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 6(6):721–741, 1984.

[57] Michaël Gharbi, Gaurav Chaurasia, Sylvain Paris, and Frédo Durand. Deep joint demosaicking and denoising. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 35(6):191:1–191:12, 2016.

[58] Michaël Gharbi, Jiawen Chen, Jonathan T Barron, Samuel W Hasinoff, and Frédo Durand. Deep bilateral learning for real-time image enhancement. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 36(4):118:1–118:12, 2017.

[59] Ralf Giering and Thomas Kaminski. Recipes for adjoint code construction. *ACM Trans. Math. Softw.*, 24(4):437–474, 1998.

[60] Mark Girolami and Ben Calderhead. Riemann manifold Langevin and Hamiltonian Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2):123–214, 2011.

[61] Ioannis Gkioulekas, Anat Levin, and Todd Zickler. An evaluation of computational imaging techniques for heterogeneous inverse scattering. In *European Conference on Computer Vision*, pages 685–701, 2016.

[62] Ioannis Gkioulekas, Shuang Zhao, Kavita Bala, Todd Zickler, and Anat Levin. Inverse volume rendering with material dictionaries. *ACM Trans. Graph.*, 32(6):162:1–162:13, nov 2013.

[63] Gabriel Goh. Why momentum really works. *Distill*, 2017.

[64] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations*, 2015.

[65] Robert Mansel Gower and MP Mello. A new framework for the computation of Hessians. *Optimization Methods and Software*, 27(2):251–273, 2012.

[66] Peter J Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, 1995.

[67] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[68] Andreas Griewank. Who invented the reverse mode of differentiation? *Documenta Mathematica, Extra Volume ISMP*, pages 389–400, 2012.

[69] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, jun 1996.

[70] Andreas Griewank and Shawn Reese. On the calculation of Jacobian matrices by the Markowitz rule. In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, 1991.

[71] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2008.

[72] Eitan Grinspun, Anil N Hirani, Mathieu Desbrun, and Peter Schröder. Discrete shells. In *Symposium on Computer Animation*, pages 62–67. Eurographics Association, 2003.

[73] Brian K Guenter. Efficient symbolic differentiation for graphics applications. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 26(3), 2007.

[74] Toshiya Hachisuka, Anton S Kaplanyan, and Carsten Dachsbacher. Multiplexed Metropolis light transport. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(4):100:1–100:10, 2014.

[75] Johannes Hanika, Anton Kaplanyan, and Carsten Dachsbacher. Improved half vector space light transport. *Comput. Graph. Forum (Proc. EGSR)*, 34(4):65–74, 2015.

[76] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234, 2016.

[77] Laurent Hascoet and Valérie Pascual. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.*, 39(3):20:1–20:43, 2013.

[78] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[79] Felix Heide, Steven Diamond, Matthias Niessner, Jonathan Ragan-Kelley, Wolfgang Heidrich, and Gordon Wetzstein. ProxImaL: Efficient image optimization using proximal algorithms. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 35(4):84:1–84:15, 2016.

[80] Felix Heide, Markus Steinberger, Yun-Ta Tsai, Mushfiqur Rouf, Dawid Pająk, Dikpal Reddy, Orazio Gallo, Jing Liu, Wolfgang Heidrich, Karen Egiazarian, Jan Kautz, and Kari Pulli. FlexISP: A flexible camera image processing framework. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(6):231:1–231:13, 2014.

[81] Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. Real-time polygonal-light shading with linearly transformed cosines. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 35(4):41:1–41:8, 2016.

[82] Eric Heitz and Stephen Hill. Linear-light shading with linearly transformed cosines. In *GPU Zen*. Black Cat Publishing, 2017.

[83] Aaron Hertzmann. Introduction to 3D non-photorealistic rendering: Silhouettes and outlines. In Stuart Green, editor, *SIGGRAPH Course Notes. Course on Non-Photorelistic Rendering*. ACM Press/ACM SIGGRAPH, New York, 1999.

[84] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In *SIGGRAPH*, pages 517–526. ACM Press/Addison-Wesley Publishing Co., 2000.

[85] Keigo Hirakawa and Thomas W. Parks. Adaptive homogeneity-directed demosaicing algorithm. *IEEE Trans. Image Process.*, 14(3):360–369, 2005.

[86] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.*, 40(4):26:1–26:16, jul 2014.

[87] Nicolas Holzschuch and FranÃğois X. Sillion. An exhaustive error-bounding algorithm for hierarchical radiosity. *Comput. Graph. Forum*, 17(4):197–218, 1998.

[88] Lars Hörmander. *The analysis of linear partial differential operators I: Distribution theory and Fourier analysis*. Springer, 1983.

[89] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial intelligence*, 17(1-3):185–203, 1981.

[90] Homan Igehy. Tracing ray differentials. In *SIGGRAPH*, pages 179–186. ACM Press/Addison-Wesley Publishing Co., 1999.

[91] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Let there be color!: joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 35(4):110:1–110:11, 2016.

[92] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox. FlowNet 2.0: Evolution of optical flow estimation with deep networks. In *Computer Vision and Pattern Recognition*. IEEE, 2017.

[93] Wolfram Research, Inc. Mathematica, Version 11.3. Champaign, IL, 2018.

[94] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. In *Advances in Neural Information Processing Systems*, pages 2017–2025, 2015.

[95] Wenzel Jakob. Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

[96] Wenzel Jakob, Miloš Hašan, Ling-Qi Yan, Jason Lawrence, Ravi Ramamoorthi, and Steve Marschner. Discrete stochastic microfacet models. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(4):115:1–115:10, 2014.

[97] Wenzel Jakob and Steve Marschner. Manifold exploration: a Markov Chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Trans. Graph.*, 31(4):58:1–58:13, 2012.

[98] Wojciech Jarosz, Derek Nowrouzezahrai, Iman Sadeghi, and Henrik Wann Jensen. A comprehensive theory of volumetric radiance estimation using photon points and beams. *ACM Trans. Graph.*, 30(1):5:1âĂŞ5:19, 2011.

[99] Wojciech Jarosz, Volker Schönefeld, Leif Kobbelt, and Henrik Wann Jensen. Theory, analysis and applications of 2D global illumination. *ACM Trans. Graph.*, 31(5):125:1–125:21, 2012.

[100] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques (Proc. EGWR)*, pages 21–30. Eurographics Association, 1996.

[101] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *International Conference on Multimedia*, MM '14, pages 675–678, 2014.

[102] Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in Neural Information Processing Systems*, pages 315–323, 2013.

[103] Michael J. Jones and Tomaso Poggio. Model-based matching by linear combinations of prototypes. Technical report, Massachusetts Institute of Technology, 1996.

[104] Nathaniel Louis Jones. *Validated interactive daylighting analysis for architectural design*. PhD thesis, Massachusetts Institute of Technology, 2017.

[105] James T. Kajiya. The rendering equation. *Comput. Graph. (Proc. SIGGRAPH)*, 20(4):143–150, 1986.

[106] Anton S Kaplanyan, Johannes Hanika, and Carsten Dachsbacher. The natural-constraint representation of the path space for efficient light transport simulation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(4):102:1–102:13, 2014.

[107] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *High Performance Graphics*, pages 33–37. ACM/Eurographics Association, 2012.

[108] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *High Performance Graphics*, pages 89–99. ACM/Eurographics Association, 2013.

[109] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3D mesh renderer. In *Computer Vision and Pattern Recognition*, pages 3907–3916. IEEE, 2018.

[110] Kenji Kawaguchi and Leslie Pack Kaelbling. Elimination of all bad local minima in deep learning. *arXiv:1901.00279*, 2019.

[111] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. A simple and robust mutation strategy for the Metropolis light transport algorithm. *Comput. Graph. Forum (Proc. Eurographics)*, 21(3):531–540, 2002.

[112] Henry J Kelley. Gradient theory of optimal flight paths. *ARS Journal*, 30(10):947–954, 1960.

[113] Markus Kettunen, Marco Manzi, Miika Aittala, Jaakko Lehtinen, Frédo Durand, and Matthias Zwicker. Gradient-domain path tracing. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 34(4):123:1–123:13, 2015.

[114] Pramook Khungurn, Daniel Schroeder, Shuang Zhao, Kavita Bala, and Steve Marschner. Matching real fabrics with micro-appearance models. *ACM Trans. Graph.*, 35(1):1:1–1:26, 2015.

[115] Diederick P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

[116] Shinya Kitaoka, Yoshifumi Kitamura, and Fumio Kishino. Replica exchange light transport. *Comput. Graph. Forum*, 28(8):2330–2342, 2009.

[117] Jaroslav Krivanek, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch. Radiance caching for efficient global illumination. *IEEE Trans. Vis. Comput. Graph.*, pages 550–561, 2005.

[118] Christopher Kulla, Alejandro Conty, Clifford Stein, and Larry Gritz. Sony Pictures Imageworks Arnold. *ACM Trans. Graph.*, 37(3):29:1–29:18, 2018.

[119] Orest Kupyn, Volodymyr Budzan, Mykola Mykhailych, Dmytro Mishkin, and Jiri Matas. DeblurGAN: Blind motion deblurring using conditional adversarial networks. *arXiv:1711.07064*, 2017.

[120] Eric P. Lafortune and Yves D. Willems. Bi-directional path tracing. In *Compugraphics*, pages 145–153, 1993.

[121] Yu-Chi Lai, , Feng Liu, and Charles Dyer. Physically-based animation rendering with Markov Chain Monte Carlo. Technical Report UW-CS-TR-1653, University of Wisconsin - Madison Computer Sciences Department, 2009.

[122] Yu-Chi Lai, Shaohua Fan, Stephen Chenney, and Charcle Dyer. Photorealistic image rendering with population Monte Carlo energy redistribution. *Rendering Techniques (Proc. EGSR)*, pages 287–295, 2007.

[123] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: wavefront path tracing on GPUs. In *High Performance Graphics*, pages 137–143. ACM/Eurographics Association, 2013.

[124] Leslie Lamport. The hyperplane method for an array computer. In *Proceedings of the Sagamore Computer Conference on Parallel Processing*, pages 113–131, 1975.

[125] Gunther Lange. Gauss type photographic objective containing two outer collective and two inner dispersive members. U.S. Patent 2,799,207 A, July 1957.

[126] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast bvh construction on gpus. *Comput. Graph. Forum (Proc. Eurographics)*, 28(2):375–384, 2009.

[127] Jaakko Lehtinen, Tero Karras, Samuli Laine, Miika Aittala, Frédo Durand, and Timo Aila. Gradient-domain Metropolis light transport. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 32(4):95:1–95:12, 2013.

[128] A Solar Lezama. *Program synthesis by sketching*. PhD thesis, UC Berkeley, 2008.

[129] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable Monte Carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[130] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 37(4):139:1–139:13, 2018.

[131] Tzu-Mao Li, Jaakko Lehtinen, Ravi Ramamoorthi, Wenzel Jakob, and Frédo Durand. Anisotropic Gaussian mutations for Metropolis light transport through Hessian-Hamiltonian dynamics. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 34(6):209:1–209:13, 2015.

[132] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's thesis, Univ. Helsinki, 1970.

[133] Guilin Liu, Duygu Ceylan, Ersin Yumer, Jimei Yang, and Jyh-Ming Lien. Material editing using a physically based rendering network. In *International Conference on Computer Vision*, pages 2280–2288. IEEE, 2017.

[134] Hsueh-Ti Derek Liu, Michael Tao, and Alec Jacobson. Paparazzi: Surface editing by way of multi-view image processing. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):221:1–221:11, 2018.

[135] Hsueh-Ti Derek Liu, Michael Tao, Chun-Liang Li, Derek Nowrouzezahrai, and Alec Jacobson. Beyond pixel norm-balls: Parametric adversaries using an analytically differentiable renderer. In *International Conference on Learning Representations*, 2019.

[136] Jun S Liu, Faming Liang, and Wing Hung Wong. The multiple-try method and local optimization in Metropolis sampling. *J. Am. Stat. Assoc.*, 95(449):121–134, 2000.

[137] Matthew M. Loper and Michael J. Black. OpenDR: An approximate differentiable renderer. In *European Conference on Computer Vision*, volume 8695, pages 154–169. ACM, 2014.

[138] Guillaume Loubet and Fabrice Neyret. Hybrid mesh-volume lods for all-scale pre-filtering of complex 3D assets. *Comput. Graph. Forum*, 36(2):431–442, 2017.

[139] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[140] Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. Deep photo style transfer. In *Computer Vision and Pattern Recognition*, pages 4990–4998. IEEE, 2017.

[141] Yi-An Ma, Yuansi Chen, Chi Jin, Nicolas Flammarion, and Michael I Jordan. Sampling can be faster than optimization. *arXiv:1811.08413*, 2018.

[142] J David MacDonald and Kellogg S Booth. Heuristics for ray tracing using space subdivision. *Visual Comput.*, 6(3):153–166, 1990.

[143] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[144] Stephan Mandt, Matthew D Hoffman, and David M Blei. Stochastic gradient descent as approximate Bayesian inference. *J. Mach. Learn. Res.*, 18(1):4873–4907, 2017.

[145] Jochem Marotzke, Ralf Giering, Kate Q Zhang, Detlef Stammer, Chris Hill, and Tong Lee. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport sensitivity. *Journal of Geophysical Research: Oceans*, 104(C12):29529–29547, 1999.

[146] James Martens. Deep learning via Hessian-free optimization. In *International Conference on Machine Learning*, volume 27, pages 735–742, 2010.

[147] James Martens, Ilya Sutskever, and Kevin Swersky. Estimating the Hessian by back-propagating curvature. In *International Conference on Machine Learning*, 2012.

[148] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 23(3):449–456, 2004.

[149] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys.*, 21(6):1087–1092, 1953.

[150] Don P Mitchell and Arun N Netravali. Reconstruction filters in computer-graphics. *Comput. Graph. (Proc. SIGGRAPH)*, 22(4):221–228, 1988.

[151] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 35(4):83:1–83:11, jul 2016.

[152] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, 2015.

[153] Uwe Naumann. *Efficient calculation of Jacobian matrices by optimized application of the chain rule to computational graphs*. PhD thesis, Verlag nicht ermittelbar, 1999.

[154] Uwe Naumann. Optimal jacobian accumulation is np-complete. *Mathematical Programming*, 112(2):427–441, 2008.

[155] Radford M. Neal. MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*, 54:113–162, 2010.

[156] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence $o(1/k^2)$. In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[157] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, second edition, 2006.

[158] Matt Olson and Hao Zhang. Silhouette extraction in Hough space. *Comput. Graph. Forum (Proc. Eurographics)*, 25(3):273–282, 2006.

[159] Hisanari Otsu, Johannes Hanika, Toshiya Hachisuka, and Carsten Dachsbacher. Geometry-aware Metropolis light transport. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):278:1–278:11, 2018.

[160] Hisanari Otsu, Anton S. Kaplanyan, Johannes Hanika, Carsten Dachsbacher, and Toshiya Hachisuka. Fusing state spaces for Markov chain Monte Carlo rendering. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 36(4):74:1–74:10, 2017.

[161] Art B. Owen. Monte Carlo theory, methods and examples, 2013.

[162] Jacopo Pantaleoni. Charted Metropolis light transport. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 36(4):75:1–75:14, 2017.

[163] Eric Paquette, Pierre Poulin, and George Drettakis. A light hierarchy for fast rendering of scenes with many lights. *Comput. Graph. Forum (Proc. Eurographics)*, pages 63–74, 1998.

[164] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 29(4):66:1–66:13, 2010.

[165] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[166] Gustavo Patow and Xavier Pueyo. A survey of inverse rendering problems. *Comput. Graph. Forum*, 22(4):663–687, 2003.

[167] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *Trans. Program. Lang. Syst.*, 30(2):7:1–7:36, 2008.

[168] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation (3rd ed.)*. Morgan Kaufmann Publishers Inc., 3rd edition, oct 2016.

[169] Matt Pharr and William R Mark. ispc: A SPMD compiler for high-performance CPU programming. In *Innovative Parallel Computing*, pages 1–13, 2012.

[170] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[171] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):32:1–32:12, jul 2012.

[172] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, 2013.

[173] Ravi Ramamoorthi, Dhruv Mahajan, and Peter Belhumeur. A first-order analysis of lighting, shading, and shadows. *ACM Trans. Graph.*, 26(1):2, 2007.

[174] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of Adam and beyond. In *International Conference on Learning Representations*, 2018.

[175] Florian Reibold, Johannes Hanika, Alisa Jung, and Carsten Dachsbacher. Selective guided sampling with complete light transport paths. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):223:1–223:14, 2018.

[176] Osborne Reynolds, Arthur William Brightmore, and William Henry Moorby. *The sub-mechanics of the universe*, volume 3. University Press, 1903.

[177] Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. A versatile scene model with differentiable visibility applied to generative pose estimation. In *International Conference on Computer Vision*, pages 765–773. IEEE, 2015.

[178] Elad Richardson, Matan Sela, Roy Or-El, and Ron Kimmel. Learning detailed face reconstruction from a single image. In *Computer Vision and Pattern Recognition*, pages 5553–5562. IEEE, 2017.

[179] Herbert Robbins and Sutton Monro. A stochastic approximation method. *Ann. Math. Statist.*, 22(3):400–407, 1951.

[180] Gareth O. Roberts and Jeffrey S. Rosenthal. Optimal scaling of discrete approximations to Langevin diffusions. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 60(1):255–268, 1998.

[181] Gareth O. Roberts and Richard L. Tweedie. Exponential convergence of Langevin distributions and their discrete approximations. *Bernoulli*, 2(4):341–363, 1996.

[182] Farbod Roosta-Khorasani and Michael W Mahoney. Sub-sampled newton methods i: globally convergent algorithms. *arXiv:1601.04737*, 2016.

[183] Farbod Roosta-Khorasani and Michael W Mahoney. Sub-sampled newton methods ii: local convergence rates. *arXiv:1601.04738*, 2016.

[184] S. Roth and M. J. Black. Fields of Experts: A framework for learning image priors. In *Computer Vision and Pattern Recognition*, volume 2, pages 860–867. IEEE, 2005.

[185] Donald B Rubin. Comment: A noniterative sampling/importance resampling alternative to the data augmentation algorithm for creating a few imputations when fractions of missing information are modest: The sir algorithm. *J. Am. Stat. Assoc.*, 82(398):543–546, 1987.

[186] Leonid I Rudin, Stanley Osher, and Emad Fatemi. Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, 60(1-4):259–268, 1992.

[187] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, pages 318–362. MIT Press, 1986.

[188] Max Sagebaum, Tim Albring, and Nicolas R. Gauger. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optimization Methods and Software*, 33(4–6):1207–1231, 2018.

[189] Pedro V. Sander, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder. Silhouette clipping. In *SIGGRAPH*, pages 327–334. ACM Press/Addison-Wesley Publishing Co., 2000.

[190] Pedro V. Sander, Hugues Hoppe, John Snyder, and Steven J. Gortler. Discontinuity edge overdraw. In *Symposium on Interactive 3D Graphics and Games*, pages 167–174. ACM, 2001.

[191] Jürgen Schmidhuber. Who invented backpropagation? `http://people.idsia.ch/~juergen/who-invented-backpropagation.html`.

[192] Mark Schmidt, Nicolas Le Roux, and Francis Bach. Minimizing finite sums with the stochastic average gradient. *Mathematical Programming*, 162(1-2):83–112, 2017.

[193] Jorge Schwarzhaupt, Henrik Wann Jensen, and Wojciech Jarosz. Practical Hessian-based error control for irradiance caching. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 31(6):193:1–193:10, 2012.

[194] Ram Shacked and Dani Lischinski. Automatic lighting design using a perceptual quality metric. *Comput. Graph. Forum*, 20(3):215–227, 2001.

[195] Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, Simon Peyton Jones, and Christoph Koch. Efficient differentiable programming in a functional array-processing language. *arXiv:1806.02136*, 2018.

[196] Mikio Shinya, T. Takahashi, and Seiichiro Naito. Principles and applications of pencil tracing. *Comput. Graph. (Proc. SIGGRAPH)*, 21(4):45–54, 1987.

[197] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[198] Savvas Sioutas, Sander Stuijk, Henk Corporaal, Twan Basten, and Lou Somers. Loop transformations leveraging hardware prefetching. In *Code Generation and Optimization*, pages 254–264. IEEE, 2018.

[199] Jascha Sohl-Dickstein, Mayur Mudigonda, and Michael DeWeese. Hamiltonian Monte Carlo without detailed balance. In *International Conference on Machine Learning*, pages 719–726, 2014.

[200] Jan Sokolowski and Jean-Paul Zolésio. Introduction to shape optimization. In *Introduction to Shape Optimization*, pages 5–12. Springer, 1992.

[201] Jos Stam. Stable fluids. In *SIGGRAPH*, pages 121–128. ACM Press/Addison-Wesley Publishing Co., 1999.

[202] Michelle Mills Strout and Paul Hovland. Linearity analysis for automatic differentiation. In *International Conference on Computational Science*, pages 574–581. Springer, 2006.

[203] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in Halide. In *Code Generation and Optimization*. IEEE, 2017.

[204] Frank Suykens and Yves D. Willems. Path differentials and applications. In *Rendering Techniques (Proc. EGWR)*, pages 257–268. Eurographics Association, 2001.

[205] Robert H Swendsen and Jian-Sheng Wang. Replica monte carlo simulation of spin-glasses. *Phys. Rev. Lett.*, 57(21):2607, 1986.

[206] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations*, 2014.

[207] Stan Development Team. *Stan Modeling Language Users Guide and Reference Manual, Version 2.9.0*, 2015.

[208] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude, 2012.

[209] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *International Conference on Computer Vision*, pages 839–846. IEEE, 1998.

[210] Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. Keyframe control of smoke simulations. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 22(3):716–723, 2003.

[211] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Deep image prior. *arXiv:1711.10925*, 2017.

[212] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Trans. Math. Softw.*, 34(4):18, 2008.

[213] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford University, 1998.

[214] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Photorealistic Rendering Techniques (Proc. EGWR)*. Eurographics Association, 1994.

[215] Eric Veach and Leonidas J. Guibas. Optimally combining sampling techniques for Monte Carlo rendering. In *SIGGRAPH*, pages 419–428. ACM Press/Addison-Wesley Publishing Co., 1995.

[216] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *SIGGRAPH*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

[217] Dominique Villard and Michael B Monagan. ADrien: an implementation of automatic differentiation in Maple. In *International Symposium on Symbolic and Algebraic Computation*, pages 221–228, 1999.

[218] Yu. M. Volin and G. M. Ostrovskii. Automatic computation of derivatives with the use of the multilevel differentiating technique — I: Algorithmic basis. *Computers and Mathematics with Applications*, 11:1099–1114, 1985.

[219] Jirí Vorba, Ondrej Karlík, Martin Sik, Tobias Ritschel, and Jaroslav Krivánek. On-line learning of parametric mixture models for light transport simulation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(4):101:1–101:11, 2014.

[220] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S Johnson, and Manfred Ernst. Embree: a kernel framework for efficient CPU ray tracing. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 33(4):143:1–143:8, 2014.

[221] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 24(3):1098–1107, 2005.

[222] Bruce Walter, Stephen R Marschner, Hongsong Li, and Kenneth E Torrance. Microfacet models for refraction through rough surfaces. *Rendering Techniques (Proc. EGSR)*, pages 195–206, 2007.

[223] Mu Wang, Alex Pothen, and Paul Hovland. Edge pushing is equivalent to vertex elimination for computing Hessians. In *Workshop on Combinatorial Scientific Computing*, pages 102–111. SIAM, 2016.

[224] Greg Ward and Paul Heckbert. Irradiance gradients. In *Eurographics Workshop on Rendering*, pages 85–98. Eurographics Association, May 1992.

[225] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *Comput. Graph. (Proc. SIGGRAPH)*, pages 85–92, 1988.

[226] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *International Conference on Machine Learning*, pages 681–688, 2011.

[227] R. E. Wengert. A simple automatic derivative evaluation program. *Commun. ACM*, 7(8):463–464, aug 1964.

[228] Paul J Werbos. Applications of advances in nonlinear sensitivity analysis. In *System modeling and optimization*, pages 762–770. Springer Berlin Heidelberg, 1982.

[229] Alexander B Wiltschko, Bart van Merriënboer, and Dan Moldovan. Tangent: automatic differentiation using source code transformation in Python. *arXiv:1711.02712*, 2017.

[230] Andrew Witkin and Michael Kass. Spacetime constraints. *Comput. Graph. (Proc. SIGGRAPH)*, 22(4):159–168, 1988.

[231] Li Xu, Jimmy Ren, Qiong Yan, Renjie Liao, and Jiaya Jia. Deep edge-aware filters. In *International Conference on Machine Learning*, pages 1669–1678, 2015.

[232] Y. Yang and C. Barnes. Approximate program smoothing using mean-variance statistics, with application to procedural shader bandlimiting. *Comput. Graph. Forum (Proc. Eurographics)*, 37(2):443–454, 2018.

[233] Yuting Yang, Sam Prestwood, and Connelly Barnes. VizGen: Accelerating visual computing prototypes in dynamic languages. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 35(6):206:1–206:13, nov 2016.

[234] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev,

Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. An introduction to computational networks and the computational network toolkit. Technical report, Microsoft Research, October 2014.

[235] Yizhou Yu, Paul Debevec, Jitendra Malik, and Tim Hawkins. Inverse global illumination: Recovering reflectance models of real scenes from photographs. In *SIGGRAPH*, pages 215–224. ACM Press/Addison-Wesley Publishing Co., 1999.

[236] Matthew D Zeiler. ADADELTA: an adaptive learning rate method. *arXiv:1212.5701*, 2012.

[237] Xiaohui Zeng, Chenxi Liu, Weichao Qiu, Lingxi Xie, Yu-Wing Tai, Chi Keung Tang, and Alan L Yuille. Adversarial attacks beyond the image space. *arXiv:1711.07183*, 2017.

[238] Chiyuan Zhang, Qianli Liao, Alexander Rakhlin, Karthik Sridharan, Brando Miranda, Noah Golowich, and Tomaso Poggio. Theory of deep learning iii: Generalization properties of sgd. Technical report, Center for Brains, Minds and Machines (CBMM), 2017.

[239] Kai Zhang, Wangmeng Zuo, Yunjin Chen, Deyu Meng, and Lei Zhang. Beyond a Gaussian denoiser: Residual learning of deep CNN for image denoising. *IEEE Trans. Image Process.*, 26(7):3142–3155, 2017.

[240] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European Conference on Computer Vision*, pages 649–666, 2016.

[241] Shuang Zhao, Frédo Durand, and Changxi Zheng. Inverse diffusion curves using shape optimization. *IEEE Trans. Vis. Comput. Graph.*, 24(7):2153–2166, 2018.

[242] Shuang Zhao, Lifan Wu, Frédo Durand, and Ravi Ramamoorthi. Downsampling scattering parameters for rendering anisotropic media. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 35(6):166:1–166:11, 2016.

[243] Henning Zimmer, Fabrice Rousselle, Wenzel Jakob, Oliver Wang, David Adler, Wojciech Jarosz, Olga Sorkine-Hornung, and Alexander Sorkine-Hornung. Path-space motion estimation and decomposition for robust animation filtering. *Comput. Graph. Forum (Proc. EGSR)*, 34(4):131–142, 2015.

[244] Barbara Zitova and Jan Flusser. Image registration methods: a survey. *Image and vision computing*, 21(11):977–1000, 2003.