SAMPLING ON METRIC GRAPHS

Rajat Vadiraj Dwaraknath & Lexing Ying
Institute for Computational and Mathematical Engineering (ICME)
Stanford University
Stanford, CA 94305
{rajatvd,lexing}@stanford.edu

ABSTRACT

Metric graphs are structures obtained by associating edges in a standard graph with segments of the real line and gluing these segments at the vertices of the graph. The resulting structure has a natural metric that allows for the study of differential operators and stochastic processes on the graph. Brownian motions in these domains have been extensively studied theoretically using their generators. However, less work has been done on practical algorithms for simulating these processes. We introduce the first algorithm for simulating Brownian motions on metric graphs through a timestep splitting Euler-Maruyama-based discretization of their corresponding stochastic differential equation. By applying this scheme to Langevin diffusions on metric graphs, we also obtain the first algorithm for sampling on metric graphs. We provide theoretical guarantees on the number of timestep splittings required for the algorithm to converge to the underlying stochastic process. We also show that the exit probabilities of the simulated particle converge to the vertex-edge jump probabilities of the underlying stochastic differential equation as the timestep goes to zero. Finally, since this method is highly parallelizable, we provide fast, memory-aware implementations of our algorithm in the form of a custom CUDA kernel that is up to $\sim$8000x faster than a GPU implementation using PyTorch. We corroborate our theoretical results with numerical experiments applying our implementation to star metric graphs. In terms of accuracy and efficiency, our scheme significantly outperforms a baseline finite volume scheme.

1 INTRODUCTION

Metric graphs, also known as quantum graphs (Kuchment, 2004), are geometric structures formed by gluing together one-dimensional segments of the real line at the vertices of an underlying graph, inheriting both the combinatorial topology of a graph and the smooth geometry of a real line. These objects have emerged as powerful tools for modeling complex systems in diverse fields, including physics, biology, and network theory. For instance, they are used to model nanoscale materials like carbon nanostructures (Amovilli et al., 2004), vascular networks (Carlson, 2006), nerve impulse transmission (Nicaise, 1985), acoustics (Cacciapuoti et al., 2006), and traffic flow on road networks (Garavello & Piccoli, 2006). We refer the reader to (Kuchment, 2002) for a comprehensive survey of the applications of quantum graphs. From a theoretical standpoint, their metric structure allows for the analysis of differential operators (Mugnolo, 2014; Erbar et al., 2022) and stochastic processes (Freidlin & Sheu, 2000), enabling the study of phenomena such as diffusion, wave propagation, and random motion on networked domains.

Brownian motions on metric graphs, a canonical example of such stochastic processes, have been extensively studied theoretically through their infinitesimal generators (Kostrykin et al., 2007; 2010; Kostrykin & Schrader, 2006; Aleandri et al., 2020). However, practical algorithms for simulating these processes – essential for numerical studies and real-world applications – have remained underdeveloped. This gap is particularly consequential in modern computational statistics and machine learning, where efficient sampling methods on complex geometries are indispensable (Byrne & Girolami, 2013; Betancourt, 2017). For example, Langevin diffusions (Roberts & Tweedie, 1996), a class of stochastic differential equations (SDEs) central to sampling from high-dimensional distributions, have seen widespread adoption in Bayesian inference (Girolami & Calderhead, 2011) and molecular dynamics (Leimkuhler & Matthews, 2015). Extending these methods to metric graphs could unlock new applications in networked systems, such as diffusive transport in dendritic networks in neuroscience (Bressloff, 2014).

Despite progress in understanding the theory of SDEs on metric graphs (Freidlin & Sheu, 2000) – including vertex transition rules, Feller properties, and large deviation asymptotics – the numerical simulation of these processes has been largely unexplored. Existing numerical work on metric graphs has focused primarily on solving partial differential equations using finite element methods (Kravitz, 2022). Some of these methods, such as finite volume schemes, struggle to stably scale to finer meshes without requiring prohibitively smaller timesteps (LeVeque, 2002) and are also less amenable to parallelization on modern hardware (GPUs) compared to Monte Carlo methods.

In this work, we bridge this gap by introducing the first algorithm (Algorithm 1) for simulating Brownian motions and Langevin diffusions on metric graphs. Our approach leverages a timestep splitting Euler-Maruyama discretization of the underlying SDE, which simultaneously resolves evolution along edges and transitions at vertices. We provide theoretical guarantees on this scheme's runtimes and consistency with the underlying SDE as the timestep approaches zero.

An important computational insight is the algorithm's parallelizability and well-suitedness to current modern GPU architectures. We implement it as a custom memory-aware CUDA kernel with Python bindings, enabling fast GPU-accelerated simulations that scale to large particle counts while effectively utilizing hardware capabilities. This implementation advances the practical utility of metric graph analyses and provides a first step toward computationally efficient stochastic simulations of these domains in high-performance computing environments. Furthermore, we demonstrate that our method significantly outperforms a baseline finite volume scheme in both accuracy and computational efficiency.

1.1 Outline

In Section 1.2, we summarize our contributions. In Section 2, we provide the necessary background on metric graphs and Brownian motions on metric graphs. In Section 3, we present our main algorithm for simulating a Brownian motion on a metric graph with implementation details in Section 3.1. In Section 4, we present numerical results on star metric graphs with drifts driven by linear and quadratic potentials.

1.2 Contributions

- We propose Algorithm 1, a timestep splitting Euler-Maruyama based discretization of the SDE of the Brownian motion, which is the first algorithm that we know of for simulating a Brownian motion and sampling on a metric graph.
- We show in Theorem 2 that the number of time-step splittings in Algorithm 1 is finite with high probability. Additionally, we show in Corollary 1 that the exit probabilities of the simulated particle using this algorithm converge to the vertex-edge jump probabilities of the underlying SDE as the timestep goes to zero.
- We provide a fast, memory-aware implementation of Algorithm 1 for GPUs in the form of a custom CUDA kernel with Python bindings and show significant speedups (up to $\sim$8000x faster) over a GPU implementation using PyTorch (Paszke et al., 2019).
- We corroborate our theoretical results with numerical experiments using our implementation on star metric graphs and significantly outperform a baseline finite volume scheme both in terms of accuracy (Figure 2) and speed (Figure 3).

Code for our implementation is available at `https://github.com/rajatvd/metric-graphs-FPI-ICLR2025`.

2 Background

2.1 Metric Graphs

In this section, we provide some formal background on metric graphs.

Definition 1 (Metric Graph)**.** *Let $G = (V, E, l)$ be an n-node, m-edge, connected, oriented graph. We associate the line segment $(0, l_e)$ with each edge $e \in E$. We identify the endpoints of the interval 0 and l_e with the corresponding vertices of the edge, which we denote e_{init} and e_{term}. The union of open metric edges associated with G is defined as $\mathbf{\Gamma}^o := \{(e, x) \mid e \in E, x \in (0, l_e)\}$, and the union of closed metric edges as $\mathbf{\Gamma}^c := \{(e, x) \mid e \in E, x \in [0, l_e]\}$. The metric graph associated with G is defined as $\mathbf{\Gamma} := V \cup \mathbf{\Gamma}^o$.*

Additionally, we allow edges to be semi-infinite, i.e., $l_e = \infty$. In this case, the terminal vertex of these edges is a vertex at infinity, and the closed intervals corresponding to these edges are $[0, \infty)$.

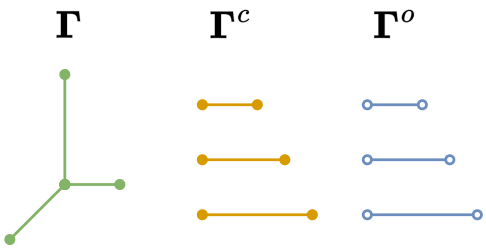

Figure 1: An example metric graph $\mathbf{\Gamma}$ and its associated spaces.

We also define a special case of metric graphs called *star metric graphs* where the graph has a single vertex and all edges are semi-infinite. The remainder of this paper will focus on star metric graphs, though all results extend to general metric graphs.

Definition 2 (Star Metric Graph)**.** *A star metric graph is a metric graph with a single vertex v, and all edges in E are semi-infinite and have length $l_e = \infty$.*

We define the set of edges incident to a vertex $v \in V$ as

$$\mathcal{E}(v) := \{e \in E \mid e_{\text{init}} = v \text{ or } e_{\text{term}} = v\}.$$

2.1.1 Function Spaces on Metric Graphs

The metric structure of each edge combined with the discrete graph metric on G leads to a natural definition of the distance $d : \mathbf{\Gamma} \times \mathbf{\Gamma} \to \mathbb{R}_+$ between two points on the metric graph. For $x, y \in \mathbf{\Gamma}$, let $\widetilde{G} = \left(\widetilde{V}, \widetilde{E}, \widetilde{l}\right)$ be the discrete graph obtained by adding two new vertices x and y to G and splitting the edges on which they lie appropriately. Then we define the distance $d(x, y)$ as the length of the shortest path between x and y in $\widetilde{G}$. This metric allows us to define the space $\mathcal{C}^k(\mathbf{\Gamma})$ as the space of functions on $\mathbf{\Gamma}$ that are k times continuously differentiable.

In addition to the global metric structure of $\mathbf{\Gamma}$, the metric structure on each edge allows us to define a broader class of continuous functions by considering continuity restricted to the edges. For a function $f : \mathbf{\Gamma} \to \mathbb{R}$ (and also for functions $f : \mathbf{\Gamma}^c \to \mathbb{R}$), we define $f_e : [0, l_e] \to \mathbb{R}$ to be the restriction of f to the closed edge e. We similarly define the restriction to open edges for functions $f : \mathbf{\Gamma}^o \to \mathbb{R}$.

We define the function space $\mathcal{C}^k(\mathbf{\Gamma}^o)$ as the space of functions on $\mathbf{\Gamma}$ whose restriction to each open edge $(0, l_e)$ is k times continuously differentiable. Note that this can be naturally extended to functions on $\mathbf{\Gamma}^c$ by extending the restrictions to have values at the endpoints of the edges as: $f_e(0) := \lim_{x \to 0^+} f_e(x)$ and $f_e(l_e) := \lim_{x \to l_e^-} f_e(x)$. By a slight abuse of notation, we will also allow the use of $f_e(e_{\text{init}}) = f_e(0)$ and $f_e(e_{\text{term}}) = f_e(l_e)$ to denote these endpoint values.

A useful observation is to note that by identification of the vertices, for two edges $e, e' \in E$ that share a vertex $v \in V$ such that $e_{\text{init}} = e'_{\text{init}} = v$, we have for $f \in \mathcal{C}(\mathbf{\Gamma})$ that

$$f_e(0) = f_{e'}(0).$$

Similar results hold for different combinations of initial and terminal vertices of the edges. However, this is not the case for functions in $\mathcal{C}^k(\mathbf{\Gamma}^c)$ or $\mathcal{C}^k(\mathbf{\Gamma}^o)$. Specifically, for $f \in \mathcal{C}^k(\mathbf{\Gamma}^c)$, it need not be the case that $f_e^{(j)}(0) = f_{e'}^{(j)}(0)$ for edges $e, e' \in E$ that share an initial vertex $v \in V$, where $f_e^{(j)}$ denotes the j-th derivative of f_e.

Finally, for notational convenience, we define the notion of an *inward derivative* along an edge at a vertex that is independent of the orientation of the edge.

Definition 3. *Let $f \in \mathcal{C}^1(\mathbf{\Gamma})$. We define the inward derivative of f at a vertex $v \in V$ along an edge $e \in E$ incident to v as*

$$\partial_e f(v) := \begin{cases} -\frac{\partial f_e}{\partial x}(0) & \text{if } e_{\text{init}} = v, \\ \frac{\partial f_e}{\partial x}(l_e) & \text{if } e_{\text{term}} = v. \end{cases}$$

Note that flipping the orientation of edge e does not change the sign of the inward derivative.

2.2 BROWNIAN MOTIONS ON METRIC GRAPHS

Brownian motions on metric graphs are extensively studied in (Kostrykin et al., 2012). A Brownian motion on a metric graph is generated by the standard second-order generator of the Brownian motion restricted to each open edge, along with specific boundary conditions at the vertices known as gluing conditions.

Let $\mu \in \mathcal{C}^1(\mathbf{\Gamma}^c)$ and $\sigma \in \mathcal{C}^1(\mathbf{\Gamma})$ be functions that denote drift and diffusion coefficients respectively. The generator $\mathcal{L}$ of a Brownian motion on the metric graph applied to a function $f \in \mathcal{C}^2(\mathbf{\Gamma})$ is given by

$$(\mathcal{L}f)_e = \mathcal{L}_e f_e \quad \text{for all } e \in E \tag{1}$$

where $\mathcal{L}_e$ is the generator of a Brownian motion on the open edge e:

$$\mathcal{L}_e f_e(x) := \frac{\partial f_e(x)}{\partial x}\mu_e(x) + \frac{1}{2}\sigma_e^2(x)\frac{\partial^2 f_e(x)}{\partial x^2}. \tag{2}$$

The domain of $\mathcal{L}$ is restricted to functions $f \in \mathcal{C}^2(\mathbf{\Gamma})$ that satisfy a set of gluing boundary conditions at each vertex $v \in V$. (Kostrykin et al., 2012) show that a class of gluing conditions called the *Wentzell boundary conditions* characterize all possible Brownian motions. The Wentzell boundary conditions for $f \in \mathcal{C}^2(\mathbf{\Gamma})$ are given by

$$a_v f(v) + \sum_{e \in \mathcal{E}(v)} b_{ve}\partial_e f(v) + \frac{1}{2}c_v f''(v) = 0 \quad \text{for all } v \in V \tag{3}$$

where $a_v \in [0,1), b_{ve} \in [0,1], c_v \in [0,1]$ are constants that satisfy

$$a_v - \sum_{e \in \mathcal{E}(v)} b_{ve} + c_v = 1 \quad \text{for all } v \in V. \tag{4}$$

In this paper, we will consider the case where $a_v = 0, c_v = 0$, which are often referred to as the *standard* boundary conditions.

Definition 4 (Standard Boundary Conditions)**.** *$f \in \mathcal{C}^2(\mathbf{\Gamma})$ satisfies the standard boundary conditions if*

$$\sum_{e \in \mathcal{E}(v)} b_{ve}\partial_e f(v) = 0 \quad \textit{for all } v \in V. \tag{5}$$

where $b_{ve} \in [0,1]$ are constants that satisfy

$$\sum_{e \in \mathcal{E}(v)} b_{ve} = 1 \quad \textit{for all } v \in V. \tag{6}$$

For convenience, we define the simplex

$$\Delta_v := \left\{ x \in \mathbb{R}^{\mathcal{E}(v)} \mid x_e \in [0,1] \text{ and } \mathbf{1}^T x = 1 \right\}$$

and note that $b_v \in \Delta_v$ defines a vertex-edge jump probability distribution at each vertex $v \in V$.

The Brownian motion generated by the generator with standard boundary conditions is conservative, and an extensive analysis of the stochastic properties is provided in (Freidlin & Sheu, 2000). In particular, (Freidlin & Sheu, 2000) derive an SDE for this Brownian motion and characterize the behavior of the process at vertices of the metric graph. As a first simplification, we only need to characterize the stochastic process's behavior at a single vertex since this is a local property that can be extended to other vertices. Effectively, we only need to consider the behavior of the process on star metric graphs. We restate the main results of (Freidlin & Sheu, 2000) in the following theorem.

Theorem 1 (Lemma 2.2 and Corollary 2.4 in (Freidlin & Sheu, 2000))**.** *Let $X_t = (e_t, x_t)$ be a Brownian motion on a star metric graph $\mathbf{\Gamma}$ with standard boundary conditions. There exists a 1-dimensional Brownian motion W_t and a local time process l_t adapted to the filtration generated by X_t such that*

$$dx_t = \mu_{e_t}(x_t)\,dt + \sigma_{e_t}(x_t)\,dW_t + dl_t. \tag{7}$$

Moreover, the local time process l_t is a continuous, non-decreasing process that only increases when the particle is at the vertex, i.e., $x_t = 0$.

Let $\tau_\delta := \inf\{t \geq 0 : x_t = \delta\}$ be the first time the process exits a ball of radius δ centered at the vertex (assume $X_0 = v$). The discrete edge process e_t is characterized by the following transition probabilities,

$$\lim_{\delta \to 0} \mathbb{P}[e_{\tau_\delta} = i] = b_{vi} \quad \textit{for all } i \in \mathcal{E}(v). \tag{8}$$

3 TIMESTEP SPLITTING EULER-MARUYAMA SCHEME FOR METRIC GRAPHS

In this section, we present our main algorithm, an Euler-Maruyama-based method for simulating Brownian motion on a metric graph via timestep splitting. First, we recall the standard Euler-Maruyama discretization for a particle on the real line $\mathbb{R}$ with the update rule

$$X_{k+1} = X_k + \mu(X_k)\,\Delta t + \sigma(X_k)\,W_{k+1}\sqrt{\Delta t}, \tag{9}$$

where W_k are i.i.d. standard normal random variables. Note that (9) is a first-order finite difference approximation with timestep Δt of the SDE

$$dX_t = \mu(X_t)\,dt + \sigma(X_t)\,dW_t, \tag{10}$$

where W_t is a standard Brownian motion.

We extend the Euler-Maruyama method to simulate Brownian motions on a metric graph $\mathbf{\Gamma}$. The main challenge in implementing a discretization scheme for the SDE (7) is to handle the case when the particle crosses a vertex in one Euler-Maruyama step in a way that is consistent with the underlying Brownian motion. To tackle this scenario, we propose a timestep splitting approach that first performs a partial Euler-Maruyama step so that the particle is exactly at the vertex and then chooses a new edge based on the vertex-edge jump probabilities b_v. Following this, we complete the remaining Euler-Maruyama step using the drift and diffusion coefficients of the new edge.

A complication that arises is that the remaining step could also result in a vertex crossing. A recursive application of the timestep splitting approach allows us to handle multiple vertex crossings in a single time step. The detailed algorithm in the case of a single vertex is described in Algorithm 1.

Algorithm 1 Timestep Splitting Euler-Maruyama Algorithm for Metric Graphs

Require: **Star** metric graph $\mathbf{\Gamma} = (V, E, l)$ (star graph so $V = \{v\}$ is a singleton and all edges are semi-infinite, with $e_{\text{init}} = v \quad \forall e \in E$), drift function $\mu : \mathbf{\Gamma}^o \to \mathbb{R}$, diffusion function $\sigma : \mathbf{\Gamma}^o \to \mathbb{R}_+$, edge-vertex jump probabilities $b_v \in \Delta_E$.

procedure EM-STEP($e, X, \Delta t$) ▷ $(e, X) \in \mathbf{\Gamma}$, Δt is the time to simulate
  $M \leftarrow 0$. ▷ Number of vertex crossings
  if $X \neq 0$ **then** ▷ Particle is not at the vertex
    Sample $W \sim \mathcal{N}(0, 1)$.
    $\widetilde{X} \leftarrow X + \mu_e(X)\,\Delta t + \sigma_e(X)\sqrt{\Delta t}W$.
    if $\widetilde{X} < 0$ **then** ▷ Particle hits vertex
      Solve $X + \alpha\mu_e(X)\,\Delta t + \sigma_e(X)\sqrt{\alpha\Delta t}W = 0$ for α.
      Sample $\widetilde{e}$ from $\mathcal{E}(v)$ according to b_v.
      $e \leftarrow \widetilde{e}$.
      $X \leftarrow 0$.
      $\Delta t \leftarrow (1 - \alpha)\Delta t$.
    else
      return $\left(e, \widetilde{X}\right)$.
  while $\Delta t > 0$ **do** ▷ Particle is at vertex
    $M \leftarrow M + 1$.
    Sample $W \sim \mathcal{N}(0, 1)$.
    $\widetilde{X} \leftarrow \widetilde{X}_1 + \mu_e(0)\,\Delta t + \sigma_e(0)\sqrt{\Delta t}\,|W|$.
    if $\widetilde{X} < 0$ **then** ▷ Particle hits vertex again
      Sample $\widetilde{e}$ from $\mathcal{E}(v)$ according to b_v.
      $e \leftarrow \widetilde{e}$.
      $\alpha \leftarrow \frac{W^2\sigma_e^2(0)}{\mu_e^2(0)\Delta t}$.
      $\Delta t \leftarrow (1 - \alpha)\Delta t$.
    else
      return $\left(e, \widetilde{X}\right)$.

A possible issue with the algorithm is that the number of timestep splittings required to simulate a single step of the Brownian motion is not guaranteed to be finite. This could lead to an infinite runtime for

simulating a finite timestep. However, we rigorously establish in Theorem 2 that this scenario does not arise, ensuring that the algorithm terminates in a finite number of steps with high probability.

Theorem 2 (Finite vertex crossings with high probability)**.** *Let M be the number of vertex crossings the particle makes in a single Euler-Maruyama step starting from the vertex, as computed in Algorithm 1 with input $(e, 0)$. Then for all $k > 0$, we have*

$$\mathbb{P}[M \leq k] \geq 1 - e^{-\frac{(k-\gamma)^2}{4k}}$$

where γ is defined as the following dimensionless quantity:

$$\gamma := \Delta t \cdot \max_{e \in \mathcal{E}(v)} \frac{\mu_e^2(v)}{\sigma_e^2(v)}.$$

In addition to the above, we also show that the exit probabilities of the simulated particle using this partial stepping algorithm converge to the vertex-edge jump probabilities of the SDE as the timestep goes to zero. Intuitively, this is because the number of vertex crossings approaches 1 as the timestep goes to zero. We formalize this in Theorem 3 and Corollary 1.

Theorem 3 (Number of crossings is 1 with high probability)**.** *Let M be the number of vertex crossings the particle makes in a single Euler-Maruyama step starting from the vertex, as computed in Algorithm 1 with input $(e, 0)$. Let γ be defined as in Theorem 2. Then,*

$$\mathbb{P}[M = 1] \geq \Omega\left(e^{-\gamma}\right).$$

As a consequence, we can choose

$$\Delta t \leq \mathcal{O}\left(\frac{1}{\max_{e \in \mathcal{E}(v)} \frac{\mu_e^2(v)}{\sigma_e^2(v)}} \log\left(\frac{1}{1-\delta}\right)\right)$$

to ensure that $\mathbb{P}[M = 1] \geq 1 - \delta$.

Corollary 1 (Jump probabilities converge to b_v)**.** *Let $(\widetilde{e}, X)$ be the output of the procedure in Algorithm 1 with timestep Δt and input $(e, 0)$. Then,*

$$\lim_{\Delta t \to 0} \mathbb{P}[\widetilde{e} = i] = b_{vi} \quad \textit{for all } i \in \mathcal{E}(v).$$

Proofs of Theorem 2, Theorem 3, and Corollary 1 can be found in Appendix A.1.

3.1 Fast, Parallelized, Memory-Aware Implementation in CUDA

The standard Euler-Maruyama discretization lends itself to a fast, parallelized implementation on GPUs because each particle can be simulated independently. Previous works have explored the use of GPUs for algorithms like Markov Chain Monte Carlo and Gibbs sampling in Euclidean spaces (Sountsov et al., 2024; Quiroz et al., 2015; Terenin et al., 2019). Our Algorithm 1 enjoys similar computational benefits, and we can leverage the parallelism of GPUs to simulate a large number of particles in parallel on metric graphs. Further, this algorithm works particularly well with GPUs' exact architecture, where the balance between memory transfers and compute operations significantly impacts practical performance.

We provide a brief overview of the architectural details of GPUs that are relevant to our implementation; further details can be found in the CUDA programming guide (Nvidia, 2011). A GPU consists of thousands of CUDA cores that can independently execute threads of computation in parallel. Along with a large number of compute units, the GPU also has a hierarchy of memory that the cores can access. The hierarchy stems from a fundamental trade-off between memory size, latency, and bandwidth. The fastest memory is the register memory, which is local to each thread and is used to store intermediate results. The next level of memory is the shared memory, which is shared between a local group of threads. The global memory is the largest and slowest memory, but it is accessible by all threads.

Achieving high performance on GPUs requires optimizing memory accesses so that the threads can maximize the utilization of the compute units. A significant advantage of Monte Carlo methods, like the standard Unadjusted Langevin Algorithm (ULA) as well as our Algorithm 1, is that they lend themselves to highly optimized memory access patterns. Specifically, since each particle simulation is completely independent,

we can assign each thread to simulate a single particle. This allows the particle's state to remain in register memory over multiple timesteps, which is the fastest memory available. Consequently, the compute units can operate at peak utilization without being bottlenecked by memory accesses. Slower transfers to and from global memory are only required when evaluating ensemble statistics, such as histogram averages. We implement our algorithm in CUDA to take advantage of these architectural features. Specifically, we provide CUDA kernels for running multiple timesteps of Algorithm 1 for multiple particles in parallel. We also provide a CUDA kernel for computing empirical histograms of these particles, which allows us to measure the error of their density with respect to the steady-state density. We present detailed numerical experiments in Section 4. CUDA kernel source code can be found in Appendix A.3.

4 Numerical Experiments

We consider a simple star metric graph with 5 edges and 1 vertex. For simplicity, we choose constant diffusion $\sigma_e(x) = \sigma$ for all edges $e \in E$. We choose the vertex-edge jump probabilities to be uniform, i.e., $b_{vi} = \frac{1}{5}$. We consider two cases of drift, driven by a linear potential and by a quadratic potential.

Linear Potential In the case of linear potentials, each edge has constant drift towards the vertex with varying magnitudes given by $\mu_{e_i}(x) = -10 \cdot i$ for $i \in \{1, 2, 3, 4, 5\}$. This drift corresponds to a linear potential with constant diffusion along each edge, which is equivalent to the Ornstein-Uhlenbeck (Ornstein, 1930) process on each edge, but edges interact through the gluing boundary conditions. The steady-state distributions ρ_i on each edge are exponential with means inversely proportional to $\frac{\mu_i}{D}$.

$$\rho_i(x) = B \exp\left(-\frac{\mu_i}{D} x\right) \quad \text{for } x \in [0, \infty], \tag{11}$$

where B is a normalization constant. Note that all edges have the same normalizing constant due to the continuity of the density at the vertex. Since the drift on each edge is constant and inward, it is not continuous at the vertex. Therefore, the gluing boundary condition (which is effectively a flux balance condition) also has a term involving the densities at the vertex. Specifically, the flux balance condition at the vertex is given by

$$\sum_{e \in E} \mu_i \rho_i(0) = D \sum_{e \in E} \partial_i \rho_i(0).$$

This condition is satisfied by the densities defined in (11) for all choices of B. Finally, the normalizing constant can be computed by setting the total mass to 1. We obtain $B = \frac{D}{\sum_i \frac{1}{\mu_i}}$.

Quadratic Potential In the case of quadratic potentials, the drift is given by $\mu_{e_i}(x) = -10 \cdot i \cdot x$ for $i \in \{1, 2, 3, 4, 5\}$. The steady-state densities in this case are Gaussians centered at the vertex with variance inversely proportional to $\frac{\mu_i}{D}$. We can use a similar argument as above to compute the normalizing constant, and we obtain the following expression for the steady state density on each edge

$$\rho_i(x) = B \exp\left(-\frac{\mu_i}{2D} x^2\right) \quad \text{for } x \in [0, \infty], \tag{12}$$

where the normalizing constant B if given by

$$B = \frac{\sqrt{\frac{2}{D\pi}}}{\sum_i \frac{1}{\sqrt{\mu_i}}}.$$

We parallelly run Algorithm 1 for multiple particles over multiple timesteps. We measure the error in the particles' density (after sufficient simulation time) with respect to the analytical steady-state density. As a baseline, we compare this with the density obtained by solving the Fokker-Planck equation using a Finite Volume Method (FVM) scheme. Numerical results are presented in Figure 2, and runtime comparisons between different implementations are in Figure 3. Further details are provided in the appendix.

5 Conclusion and Future Work

In this paper, we presented a novel Euler-Maruyama based algorithm for simulating Brownian motions on metric graphs. Our algorithm uses a timestep splitting approach that allows us to handle vertex crossings in

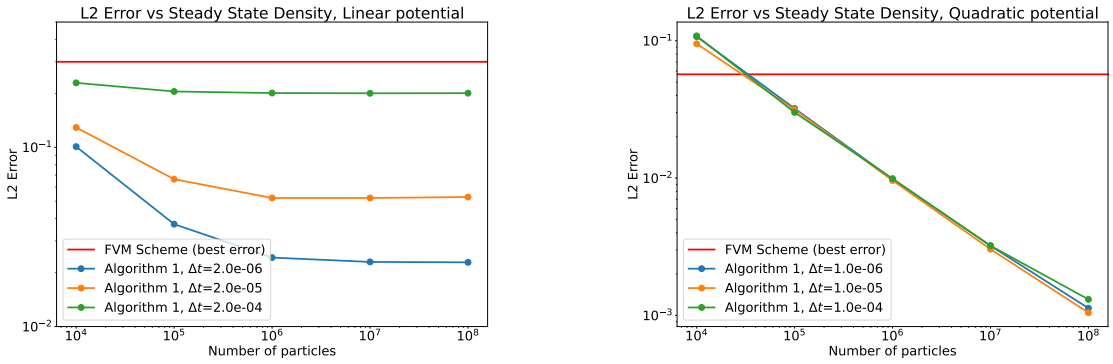

Figure 2: Error in density estimation for linear and quadratic potentials. The FVM scheme directly solves the Fokker-Planck equation to obtain the steady-state density. We compare the *best case* error (over discretization parameters) of this scheme with the error obtained by running Algorithm 1 for multiple particle counts and values of the timestep. We estimate the density using a simple histogram with a bin size equal to the discretization of the FVM scheme. The error is computed as the empirical L2 distance between the estimated density and the analytical steady-state density. We observe that Algorithm 1 results in significantly lower error compared to the FVM scheme for the same level of spatial and time discretizations.

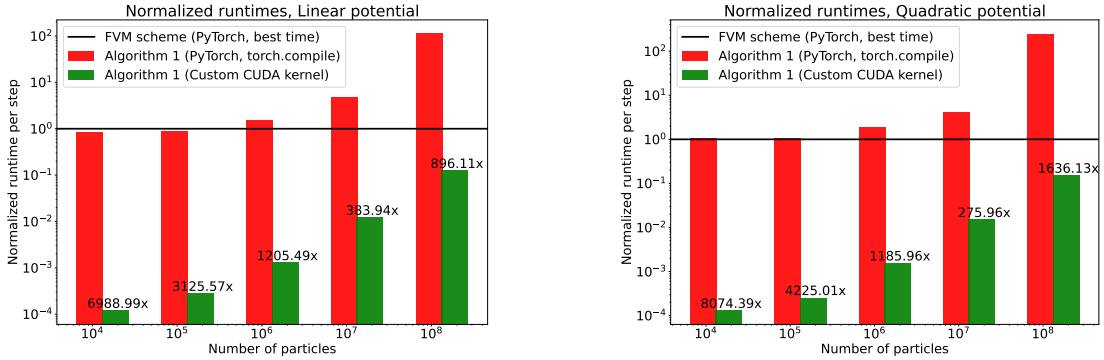

Figure 3: Normalized runtimes per step, aggregated over different discretization parameters for Algorithm 1 for linear and quadratic potentials compared with the best runtime for the FVM scheme. We observe that the FVM scheme has a significantly higher runtime compared to Algorithm 1 for the same level of spatial and time discretizations. Additionally, our custom CUDA kernel for Algorithm 1 is significantly faster (up to $\sim$8000x speedup) than the PyTorch implementation (speedups indicated on the bars). We observe slightly higher runtimes for the linear potential, which is expected due to the increased likelihood of vertex crossings per timestep. All experiments were run on an NVIDIA RTX A6000 GPU.

a way that is consistent with the underlying Brownian motion. We rigorously established that the number of vertex crossings is finite with high probability and that the exit probabilities of the simulated particle converge to the vertex-edge jump probabilities of the SDE as the timestep goes to zero. We also provided a fast, parallelized, memory-aware implementation in CUDA that takes advantage of the architecture of modern GPUs. We demonstrated the effectiveness of our algorithm through numerical experiments on a simple star metric graph with linear and quadratic potentials.

Promising future directions include developing higher order variants of timestep splitting algorithms like Algorithm 1 for simulating Brownian motions on metric graphs. Further, bringing existing sampling algorithms inspired by optimization perspectives (Chewi, 2023) like proximal sampling (Liang & Chen, 2022), mirror Langevin (Hsieh et al., 2018), and the Metropolis-adjusted Langevin algorithm (Xifara et al., 2014) to the domain of metric graphs would serve as an interesting research direction. On the theoretical side, developing non-asymptotic convergence rates for these algorithms is a potential avenue for future work. Finally, extending the optimized CUDA implementation to accommodate interacting systems by exploiting shared memory and other architectural features of GPUs is another promising direction.

ACKNOWLEDGMENTS

The authors would like to thank Henrique Monteiro for helpful discussions about Algorithm 1.

REFERENCES

Michele Aleandri, Matteo Colangeli, and Davide Gabrielli. A combinatorial representation for the invariant measure of diffusion processes on metric graphs. *arXiv preprint arXiv:2002.00654*, 2020.

Claudio Amovilli, Frederik E Leys, and Norman H March. Electronic energy spectrum of two-dimensional solids and a chain of c atoms from a quantum network model. *Journal of mathematical chemistry*, 36: 93–112, 2004.

Michael Betancourt. A conceptual introduction to hamiltonian monte carlo. *arXiv preprint arXiv:1701.02434*, 2017.

Paul C Bressloff. *Stochastic processes in cell biology*, volume 41. Springer, 2014.

Simon Byrne and Mark Girolami. Geodesic monte carlo on embedded manifolds. *Scandinavian Journal of Statistics*, 40(4):825–845, 2013.

Carmela Cacciapuoti, Rodolfo Figari, and Andrea Posilicano. Point interactions in acoustics: One-dimensional models. *Journal of mathematical physics*, 47(6), 2006.

Robert Carlson. Linear network models related to blood flow. *Contemporary Mathematics*, 415:65–80, 2006.

George Casella. L.; berger, rl statistical inference, 2001.

Sinho Chewi. *An optimization perspective on log-concave sampling and beyond*. PhD thesis, Massachusetts Institute of Technology, 2023.

Matthias Erbar, Dominik Forkert, Jan Maas, and Delio Mugnolo. Gradient flow formulation of diffusion equations in the Wasserstein space over a Metric graph. *Networks and Heterogeneous Media*, 17(5):687, 2022. ISSN 1556-1801, 1556-181X. doi: 10.3934/nhm.2022023. URL https://www.aimsciences.org/article/doi/10.3934/nhm.2022023.

Mark Freidlin and Shuenn-Jyi Sheu. Diffusion processes on graphs: stochastic differential equations, large deviation principle. *Probability Theory and Related Fields*, 116(2):181–220, February 2000. ISSN 1432-2064. doi: 10.1007/PL00008726. URL https://doi.org/10.1007/PL00008726.

Mauro Garavello and Benedetto Piccoli. Traffic flow on a road network using the aw–rascle model. *Communications in Partial Differential Equations*, 31(2):243–275, 2006.

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.

Ya-Ping Hsieh, Ali Kavis, Paul Rolland, and Volkan Cevher. Mirrored langevin dynamics. *Advances in Neural Information Processing Systems*, 31, 2018.

Vadim Kostrykin and Robert Schrader. Laplacians on Metric Graphs: Eigenvalues, Resolvents and Semigroups, January 2006. URL http://arxiv.org/abs/math-ph/0601041. arXiv:math-ph/0601041.

Vadim Kostrykin, Jurgen Potthoff, and Robert Schrader. Heat kernels on metric graphs and a trace formula, January 2007. URL http://arxiv.org/abs/math-ph/0701009. arXiv:math-ph/0701009.

Vadim Kostrykin, Jurgen Potthoff, and Robert Schrader. Brownian Motions on Metric Graphs: Feller Brownian Motions on Intervals Revisited, August 2010. URL https://arxiv.org/abs/1008.3761v2.

Vadim Kostrykin, Jürgen Potthoff, and Robert Schrader. Brownian Motions on Metric Graphs. *Journal of Mathematical Physics*, 53(9):095206, September 2012. ISSN 0022-2488, 1089-7658. doi: 10.1063/1.4714661. URL http://arxiv.org/abs/1102.4937. arXiv:1102.4937 [math].

Hannah Kravitz. *Metric Graphs: Numerical Methods, Localization, and the Spread of Epidemics*. Ph.D., The University of Arizona, United States – Arizona, 2022. URL https://www.proquest.com/docview/2708229288/abstract/7AEE8B405BEA4340PQ/1. ISBN: 9798841733942.

P Kuchment. Graph models of wave propagation in thin structures, waves random media 12. *R1-R24*, 2002.

Peter Kuchment. Quantum graphs: I. Some basic structures. *Waves in Random Media*, 14 (1):S107–S128, January 2004. ISSN 0959-7174, 1361-6676. doi: 10.1088/0959-7174/14/1/014. URL http://www.informaworld.com/openurl?genre=article&doi=10.1088/0959-7174/14/1/014&magic=crossref||D404A21C5BB053405B1A640AFFD44AE3.

Beatrice Laurent and Pascal Massart. Adaptive estimation of a quadratic functional by model selection. *Annals of statistics*, pp. 1302–1338, 2000.

Ben Leimkuhler and Charles Matthews. Molecular dynamics. *Interdisciplinary applied mathematics*, 39 (1), 2015.

Randall J LeVeque. *Finite volume methods for hyperbolic problems*, volume 31. Cambridge university press, 2002.

Jiaming Liang and Yongxin Chen. A proximal algorithm for sampling. *arXiv preprint arXiv:2202.13975*, 2022.

Delio Mugnolo. *Semigroup methods for evolution equations on networks*, volume 20. Springer, 2014.

Serge Nicaise. Some results on spectral theory over networks, applied to nerve impulse transmission. In *Polynômes Orthogonaux et Applications: Proceedings of the Laguerre Symposium held at Bar-le-Duc, October 15–18, 1984*, pp. 532–541. Springer, 1985.

CUDA Nvidia. Nvidia cuda c programming guide. *Nvidia Corporation*, 120(18):8, 2011.

Leonard Salomon Ornstein. On the theory of the brownian motion. *Physical review*, 36:823–841, 1930.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Matias Quiroz, Mattias Villani, and Robert Kohn. Scalable mcmc for large data problems using data subsampling and the difference estimator. 2015.

Gareth O Roberts and Richard L Tweedie. Exponential convergence of langevin distributions and their discrete approximations. 1996.

Pavel Sountsov, Colin Carroll, and Matthew D Hoffman. Running markov chain monte carlo on modern hardware and software. *arXiv preprint arXiv:2411.04260*, 2024.

Alexander Terenin, Shawfeng Dong, and David Draper. Gpu-accelerated gibbs sampling: a case study of the horseshoe probit model. *Statistics and computing*, 29:301–310, 2019.

Tatiana Xifara, Chris Sherlock, Samuel Livingstone, Simon Byrne, and Mark Girolami. Langevin diffusions and the metropolis-adjusted langevin algorithm. *Statistics & Probability Letters*, 91:14–19, 2014.

A Appendix

A.1 Proofs of Theorems

A.1.1 Finite Vertex Crossings with High Probability

Proof of Theorem 2. Let I_k for $k > 0$ be iid variables that take values in $\mathcal{E}(v)$ according to the distribution b_v. Let $W_k \sim \mathcal{N}(0, 1)$ for $k > 0$. We define the following sequence of random variables:

$$h_k := h_{k-1} - \frac{\sigma^2_{I_k}(v)}{\mu^2_{I_k}(v)} W_k^2,$$

where $h_0 := \Delta t$.

Define the stopping time $\tau := \inf\{k > 0 : h_k \leq 0\}$. First, observe that $M = \tau$. To see this, note that after one iteration of the loop in Algorithm 1, we have $\Delta t' = \Delta t - \frac{\sigma_{I_1}^2(v)}{\mu_{I_1}^2(v)} W_1^2$. If $\Delta t' \leq 0$, then $M = 1$, and if $\Delta t' > 0$, then we continue the loop with $\Delta t'$ in place of Δt. By induction, we see that $M = \tau$. We can define the following upper bounding sequence by considering the worst case over all possible choices of I_k,

$$\widetilde{h}_k := \widetilde{h}_{k-1} - \frac{h_0}{\gamma} W_k^2, \text{ where } \widetilde{h}_0 = h_0 = \Delta t.$$

We can solve the recursion for $\widetilde{h}_k$ to get

$$\widetilde{h}_k = h_0 \left(1 - \frac{1}{\gamma} \sum_{i=1}^{k} W_i^2\right).$$

By construction, $\widetilde{h}_k \geq h_k$ for all $k \geq 0$. We define a similar stopping time for $\widetilde{h}$, as $\widetilde{\tau} := \inf\left\{k > 0 : \widetilde{h}_k \leq 0\right\}$. Clearly, $h_k \leq \widetilde{h}_k \implies \widetilde{\tau} \geq \tau$. Now, observe that $\sum_{i=1}^{k} W_i^2 = \chi_k^2$ is a chi-squared random variable with k degrees of freedom. We have,

$$h_0 \left(1 - \frac{1}{\gamma}\chi_k^2\right) \leq 0 \iff \chi_k^2 \geq \gamma \implies \widetilde{\tau} \leq k \implies \tau \leq k.$$

Therefore,

$$\mathbb{P}\left[M \leq k\right] = \mathbb{P}\left[\tau \leq k\right] \geq \mathbb{P}\left[\widetilde{\tau} \leq k\right] \geq \mathbb{P}\left[\chi_k^2 \geq \gamma\right].$$

To control the tail of χ_k^2, we use the following bound from Lemma 1 in Section 4.1 of (Laurent & Massart, 2000),

$$\mathbb{P}\left[k - \chi_k^2 \geq 2\sqrt{kx}\right] \leq e^{-x}.$$

We have,

$$\begin{aligned} \mathbb{P}\left[\chi_k^2 \leq k - 2\sqrt{kx}\right] &\leq e^{-x} \\ \implies 1 - \mathbb{P}\left[\chi_k^2 \geq k - 2\sqrt{kx}\right] &\leq e^{-x}. \end{aligned}$$

By setting $x = \frac{(k-\gamma)^2}{4k}$, we get

$$\mathbb{P}\left[\chi_k^2 \geq \gamma\right] \geq 1 - e^{-\frac{(k-\gamma)^2}{4k}}.$$

This completes the proof. □

A.1.2 Number of Crossings is 1 with High Probability

Proof of Theorem 3. We use the same notation defined in the proof of Theorem 2.

First, observe that $M = 1 \implies W_1^2 \geq \gamma$.

So, we have $\mathbb{P}\left[M = 1\right] \leq \mathbb{P}\left[W_1^2 \geq \gamma\right]$.

Since W_1 is a standard normal random variable, we have

$$\mathbb{P}\left[W_1^2 \geq \gamma\right] = \mathbb{P}\left[|W_1| \geq \sqrt{\gamma}\right] = 2\left(1 - \Phi\left(\sqrt{\gamma}\right)\right),$$

where Φ is the CDF of the standard normal distribution.

We use a standard lower bound on the CDF (Casella, 2001) of the normal distribution to get

$$\begin{aligned} 1 - \Phi\left(\sqrt{\gamma}\right) &\geq C \exp\left(-\frac{\gamma}{2}\right) \\ \implies \mathbb{P}\left[M = 1\right] &\geq C \exp\left(-\frac{\gamma}{2}\right) \geq \Omega\left(e^{-\gamma}\right). \end{aligned}$$

This completes the proof.

□

A.1.3 JUMP PROBABILITIES CONVERGE TO b_v

Proof of Corollary 1. Clearly, if $M = 1$, then $\widetilde{e} = i$ with probability b_{vi}, since $\widetilde{e}$ is sampled from the distribution b_v once. By noting that $\gamma \to 0$ as $\Delta t \to 0$, we can apply Theorem 3 to conclude that $\mathbb{P}[M = 1] \to 1$ as $\Delta t \to 0$. This completes the proof. □

A.2 BASELINE FINITE VOLUME SCHEME

We provide a brief overview of the Finite Volume Method (FVM) scheme for solving the Fokker-Planck equation on metric graphs. On each edge $e \in E$, we discretize the Fokker-Planck equation using a standard FVM scheme; see (LeVeque, 2002) for details. We use an upwinding scheme to discretize the drift term and a central difference scheme to discretize the diffusion term. We use a first-order explicit Euler scheme to discretize the time derivative.

We now provide details of the flux balance condition at the vertex. Denote the density of the cell adjacent to the vertex on edge i by ρ_i. To mimic the gluing boundary conditions, we use a flux distribution at the vertex that is proportional to the jump probabilities b_v. Specifically, let F_{ij} be the flux from edge i to edge j at the vertex. We decompose the flux into a drift and diffusion component.

Drift Component Since we use an upwinding scheme to discretize the drift term, the drift component of the flux from edge i to edge j is zero if the drift $\mu_i(v)$ is away from the vertex on edge i. If the drift is towards the vertex, then the drift component of the flux is given by

$$F_{ij}^{\text{drift}} = \mu_i(v)\rho_i / b_{vi} \frac{b_{vj}}{\sum_{k \neq i} b_{vk}}.$$

Note that the drift flux is distributed to all target edges, proportional to the jump probabilities. The density from the source is normalized by the jump probability to account for the fact that the density is distributed to all target edges.

Intuitively, the jump probability can be interpreted as the relative "cross-sectional areas" of the edges at the vertex. The density is the linear density of the particles on the edge. When computing fluxes across different edges, we need to account for the relative "cross-sectional areas" of the edges at the vertex. Hence, we normalize by the appropriate jump probabilities.

Diffusion Component The diffusion component of the flux is given by

$$F_{ij}^{\text{diffusion}} = \frac{\sigma^(v)}{2} \left(\frac{\rho_i / b_{vi} - \rho_j / b_{vj}}{\Delta x} \right) b_{vj}$$

A similar normalization is applied to the density terms to account for the relative "cross-sectional areas" of the edges at the vertex.

The total flux into the cells at the vertex on each edge is the sum of all the incoming drift and diffusion components from every other edge.

A.3 CUDA KERNELS

We present source code for our CUDA kernels for running Algorithm 1 for multiple particles over multiple timesteps. Python bindings and other code can be found in the repository at `https://github.com/rajatvd/metric-graphs-FPI-ICLR2025`.

```
#include <curand_kernel.h>

#define TOL 1e-10f
#define MAX_ITERATIONS_PER_STEP 100
#define STEPS_PER_KERNEL 1000

extern "C" {
__device__ float device_dV(int edge_index, float x) {
  // gradient of quadratic potential
```

```
  return 10.0f * (edge_index + 1.0f) * x;
}

__device__ float solve_quadratic(float A, float B, float C) {
  // Numerically stable solution to quadratic equation
  if (A == 0.0f) {
    return -C / B;
  }
  float discriminant = sqrtf(fmaxf(B * B - 4.0f * A * C, 0.0f));
  if (B > 0.0f) {
    return (-B - discriminant) / (2.0f * A);
  } else {
    return (2.0f * C) / (-B + discriminant);
  }
}

__global__ void
langevin_multi_step_kernel(int *edges, float *positions, int *bounces,
                           int *bounce_instances, const float *edge_lengths,
                           const float *jump_weights, const float base_dt,
                           const float sigma, const int num_edges,
                           const int num_particles, curandState *states) {
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid >= num_particles)
    return;

  int edge = edges[tid];
  float x = positions[tid];
  int bounce_count = bounces[tid];
  int bounce_instance = bounce_instances[tid];
  curandState local_state = states[tid];

  if (edge < 0 || edge >= num_edges) {
    printf("Invalid initial edge %d for particle %d\n", edge, tid);
    edge = 0;
  }

  for (int step = 0; step < STEPS_PER_KERNEL; ++step) {
    float dt = base_dt;
    int iterations = 0;

    while (dt > 0.0f && iterations++ < MAX_ITERATIONS_PER_STEP) {
      float w = curand_normal(&local_state);
      if (x == 0.0f)
        w = fabsf(w);

      float drift = -device_dV(edge, x);
      float sqrt_dt = sqrtf(dt);
      float x_next = x + dt * drift + sigma * sqrt_dt * w;
      float current_length = edge_lengths[edge];

      if (current_length <= 0.0f) {
        printf("Invalid edge length %f for edge %d\n", current_length, edge);
        current_length = 1.0f;
      }

      if (x_next > 0.0f && x_next <= current_length) {
        // no bounce
```

```
        x = x_next;
        dt = 0.0f;
      } else if (x_next > current_length) {
        // also no bounce
        x = 2.0f * current_length - x_next;
        dt = 0.0f;
      } else {
        // x_next < 0.0f -- bounce
        if (x != 0.0f) {
          // first bounce
          bounce_instance++;
        }
        bounce_count++;

        float a = drift * dt;
        float b = sigma * sqrt_dt * w;
        float sqrt_alpha = solve_quadratic(a, b, x);
        float alpha = sqrt_alpha * sqrt_alpha;

        dt *= (1.0f - alpha);
        float rand_val = curand_uniform(&local_state);
        int new_edge = 0;
        while (new_edge < num_edges - 1 && rand_val > jump_weights[new_edge]) {
          new_edge++;
        }

        if (new_edge < 0 || new_edge >= num_edges) {
          printf("Invalid new_edge %d, clamping to 0\n", new_edge);
          new_edge = 0;
        }

        edge = new_edge;
        x = 0.0f;
      }
    }
  }

  edges[tid] = edge;
  positions[tid] = x;
  bounces[tid] = bounce_count;
  bounce_instances[tid] = bounce_instance;
  states[tid] = local_state;
}

__global__ void setup_kernel(curandState *states, unsigned long long seed,
                             int num_particles) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid >= num_particles)
    return;
  curand_init(seed + tid, 0, 0, &states[tid]);
}

// histogram kernel
__global__ void compute_histogram_kernel(const int *edges,
                                         const float *positions,
                                         const float *edge_lengths,
                                         int *histograms, int num_bins,
                                         int num_edges, int num_particles) {
```

```
  const int tid = blockIdx.x * blockDim.x + threadIdx.x;
  if (tid >= num_particles)
    return;

  const int edge = edges[tid];
  const float pos = positions[tid];

  // Validate input
  if (edge < 0 || edge >= num_edges)
    return;
  const float length = edge_lengths[edge];
  if (pos < 0.0f || pos > length)
    return;

  // Calculate normalized position [0,1]
  const float normalized_pos = pos / length;

  // Determine bin index
  int bin = (int)(normalized_pos * num_bins);
  bin = max(0, min(bin, num_bins - 1)); // Clamp to valid range

  // Atomic increment using 2D indexing: [edge][bin]
  atomicAdd(&histograms[edge * num_bins + bin], 1);
}
}
```