Computational Geometry: Querying Particle Dynamics

Theodore Wu
wut6@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

William Allen
allenw@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York, USA

ABSTRACT

We propose a novel segment tree-based data structure which can answer particle position queries in efficient time across timestep ranges. The proposed algorithm is capable of answering queries at fractional timesteps by interpolating between the nearest measured location for each particle. Additionally, we implement three types of particle simulations using computational geometry techniques to test our data structure. We discuss the theoretical optimal running time, and why an approach using kinetic data structures cannot improve the memory required to store the required data.

1 INTRODUCTION

The field of particle simulation design is deep and well studied. We explore a general approach for storing and querying particle positions retrospectively, with no knowledge of the queries to be executed prior to the completion of the simulation. This approach is useful for time-stepped simulations where the state at each timestep is expensive to compute, but the total number of particles is small. This approach can also prove useful for a distributed system where each particle records its own position over time, and sends the full path to a central location at the end of a longer time period, such as a network of sensors tracking tectonic activity, or the movement of glaciers.

2 BACKGROUND

The first simulation we made was based on a discretization of Ficks law. Ficks law relates the change in concentration to the flux as : $J = -D\frac{d\phi}{dx}$. Where J is the diffusion flux or the amount of unit substance per unit area per unit time. ϕ is the concentration or the amount of substance per unit volume. D is the diffusion coefficient. If we discretize space and time, we can simulate the concentration of some substance, given the diffusion flux and the diffusion coefficient.

The discretization of space in 2D, is shown in Figure 1. Any concentration in the middle square will move to any adjacent square as time progresses. Time is discretized by having the simulation advance in discrete timesteps.

A segment tree is a spatial data structure which allows efficient querying of the line segments which intersect an arbitrary query window. It has been shown that for a dD segment tree, a query can be executed in $O(\log^d(n) + k)$[2] time, where d is the number of dimensions, and k is the output size.

3 SIMULATION DESIGN

We created three types of simulations for this project. A coarse simulation based on Fick's Law, a simulation of the Wierner process and a simulation of elastic collisions.

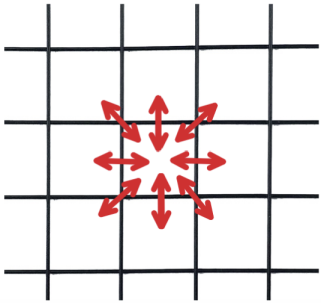

Figure 1: A visualization of how the concentration in the middle square will change over time

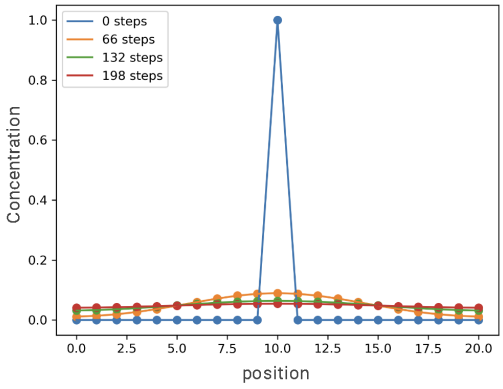

Figure 2: Concentration at each position for the timesteps specified

The first simulation models Fick's law and was made using the python Scipy library. Specifically the convolve2D function in the signal package. The change in concentration in one unit timestep across one unit length can be modeled with a convolution. The convolution between the entire simulation space and another 3x3 matrix representing the diffusion flux divided by the diffusion coefficient. Scipy allows for boundary conditions to be set, which align nicely with what behavior should occur at simulation boundaries. It allows for concentration to be reflected across the boundaries (so that the overall concentration of the simulation will remain constant), or for the concentration to leave the simulation and pass through the boundary of the simulation.

The results of this simulation can be summarized with the graph in Figure 2. The simulation is initialized as a 1D line and starts with a single unit of concentration in the middle. Notice that the concentration will become uniform across the entire simulation over time.

To set up this simulation, one would need the diffusion flux, the size of the entire simulation and the initial concentrations. These are provided as numpy matrices. To advance the simulation by one timestep, a user would need to provide the amount of real time that has passed. To simulate Fick's law we can simply apply the diffusion flux over the entire matrix of concentrations to get the change in concentrations. This can then be multiplied by a percentage of the unit time (to use smaller timesteps). The change in concentration can then be added to the current concentration matrix to get the next timestep.

Our query algorithm should be able to show similar results as this simulation. The overall effect of many individual particles is summarized by Ficks law and having this simulation can be used as a way to verify that.

The next two simulations we made with the intent to be able to create particle data for our query algorithm. The second simulation generates data through the Wierner process. The random motion of particles suspending in a medium, is called Brownian Motion. The Wierner process is one way that Brownian Motion can be simulated. The Wierner process can be described as a continuous time stochastic process. For times t and s where $0 \leq s \leq t$ the following must hold: $W_t - W_s \sim \mathcal{N}(0, s-t)$. Using the central limit theorem it can be shown that this process can be approximated as a random walk [3]. To simulate this process for each particle we just need to be able to generate random numbers from a Gaussian distribution. The position for each particle at timestep t can be calculated according to the following formula: $W_n(t) = \frac{1}{\sqrt{n}} \sum_{1 \leq k \leq \lfloor nt \rfloor} \epsilon_k$ for $\epsilon_k \sim \mathcal{N}(0,1)$. This process will approach Brownian motion as $n \to \infty$. Our simulation uses $n = 10$, but this could easily be set to a higher number for greater precision. For each timestep t, we only need to calculate $W_n(t) - W_n(t-1)$ to get the updated position of each particle. This is calculated for both dimensions for every particle. The result of this simulation is the position of each particle at every timestep, which can then be given to our query algorithm.

The final simulation, involves elastic collisions. In kinetic molecular theory, collisions between gas particles or collisions with the walls of the container are perfectly elastic. None of the energy of a gas particle is lost when it collides with another particle or with the walls of the container. This means we can model each particle's interactions with its neighbors and the surrounding environment in accordance with the formulas that govern elastic collisions. These formulas utilize the conservation of momentum to determine the velocities of each particle before and after collisions.

Each particle is determined by it's position and velocity. At each timestep, a new position and velocity is calculated in accordance with the following process. First the particle is moved based on its current velocity for Δt amount of time. Then collisions between particles and the boundary are checked. If the simulation is set to have a reflective boundary, the velocity and position of the particle is resolved, to have the particle bounce off of the boundary. Lastly, particle to particle collisions are calculated. If two particles overlap, their positions and velocities are resolved in accordance with an elastic collision [4].

Collisions between particles are checked using a Delaunay graph. At every timestep the simulation need to determine if two particles have collided. There are $O(n^2)$ possible pairs of particles that can collide, but it would be infeasible to do this at every timestep. The way I check for collisions is to first construct the Delaunay graph using the Scipy library. Then collisions are only checked between particles if there is an edge between particles in the Delaunay graph. It takes $O(nlogn)$ to construct the Delaunay graph, so this represents a significant speedup.

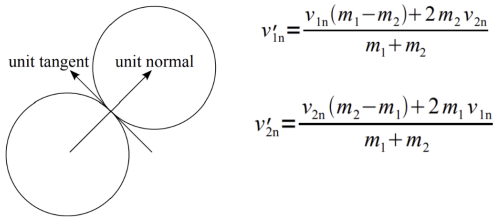

Figure 3: The moment of collision between two particles. v_{1n} is velocity of particle 1 projected onto the unit normal. v'_n particles velocity after the collision.

Resolving each particle's velocity and position involves calculating the unit tangent and unit normal for each collision [4]. If two particles have collided, the distance between their centers is less than the sum of their radii. If they have collided, we can determine the exact time they had collided, by considering their current velocity. As shown in Figure 3, given two particles that have collided, only the normal component of their velocities will change. The tangential components of their velocities will remain the same.

The pseudocode for these simulations is as follows:

Algorithm 1 Progress the simulation by time dt

```
function STEP(SIMULATIONDATA, dt)
    simulationData ← Each particles position after progressing time by dt (Either by their velocity or something else)
    for particle in simulationData do
        if particle has crossed the simulation boundary then
            Reflect particle off boundary
    /* This next part is only need for collision checking */
    delaunay ← delaunay_graph(simulationData)
    for edge in delaunay do
        p1 ← edge.particle1
        p2 ← edge.particle2
        if length of edge ≤ radius(p1) + radius (p2) then
            Resolve collisions between p1 and p2 using [4]
    return simulationData
```

4 QUERY ALGORITHM

Given a set of particles in motion, whose positions are recorded across a series of timesteps, we wish to query the position of particles within arbitrary bounding boxes and timestep ranges. The most general approach to this problem is to create a data structure which stores the location of each point at each timestep, and only allow queries at discrete timesteps. Such a general approach leads to the potential for particles to pass undetected through a query window between timesteps, without ever being explicitly observed within it,

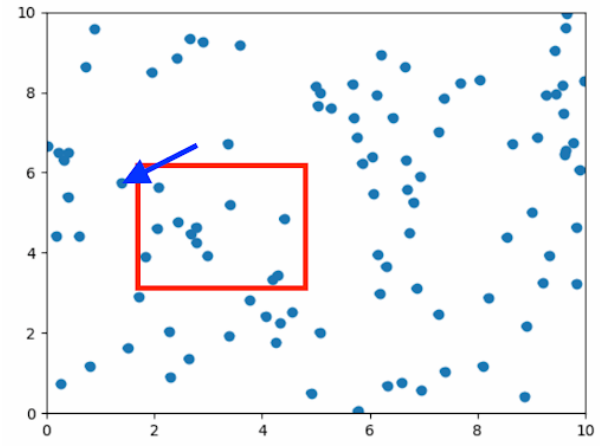

Figure 4: A query window, with an arrow indicating the path of a particle between two timesteps. The particle is never observed inside the query window, but we can infer that it passed through.

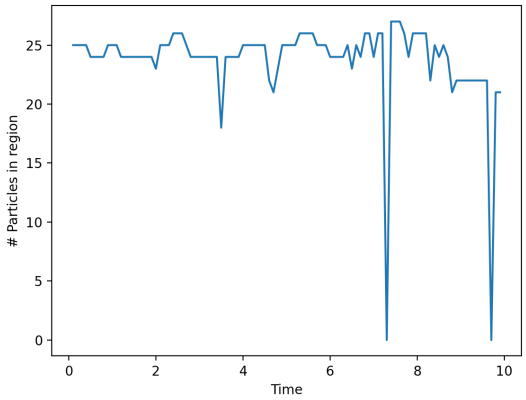

Figure 5: Density of a query region for a simulation based on the Wierner process, with 100 timesteps and 500 particles. The simulation was confined to a 10x10 simulation region, with a 2x2 query window at the center of the simulation.

as shown in Figure 4 Our proposed query algorithm mitigates this concern by instead storing line segments between each observed position for a particle, as an approximation of the path a particle traveled between timesteps.

Without any better information about the path traveled by a particle between two timesteps, we can linearly interpolate between positions to gain a rough estimate of the path traveled in the intervening time. This works reasonably well for sparse simulations where particles travel long distances between timesteps and rarely interact with one another, but it breaks down when the simulation is dense. Dense simulations with a high number of collisions, and simulations with high particle velocities have greater error than sparser, less energized, simulations. Our query algorithm is most effective for sparse simulations with relatively few timesteps, where an average particle movement from one timestep to another contains no collisions.

To perform efficient window queries across a timestep range, we store simulation data as an array of segment trees, one per timestep, where each segment tree contains line segments representing the movement of particles between the previous timestep to the current timestep. Since the timesteps are evenly spaced, and are strictly increasing over time, it is more efficient to store the first level of the data structure as an array whose indices can be accessed in constant time. To handle fractional timesteps, we query all of the timesteps which overlap with the fractional timestep range, and then perform a linear scan to filter out those segments which are inferred to have missed the query region during the requested time interval. The data structure is demonstrated in Figure 5 , where we plot the density of a query region over time for the duration of a 100-timestep simulation with 500 particles.

Our segment tree implementation follows that of Berg, et. al.[1]. Each segment tree takes $O(n * \log(n))$ memory, while the entire data structure consists of t segment trees, and thus takes $O(n * \log(n) * t)$ memory. As discussed in greater detail in the following paragraph, the worst-case lower bound for the total memory usage of a data structure capable of storing the state of a simulation at every timestep is in $O(n * t)$, for n particles and t timesteps. The extra factor of $O(\log(n))$ total memory usage allows us to answer window queries in efficient $O(\frac{k}{n} \log^2(n) + k)$ time, for n particles, and k total results.

One of our stated goals was to create a data structure which could efficiently answer queries about particle positions without using $O(n * t)$ memory, for n particles and t timesteps. In hindsight, the proof of the futility of this concept is simple: given a system with an set of particles initialized with random positions and velocities, the entropy of the system is conserved, and the amount of information at any given timestep is equal to that any any other timestep. What is not constant is the amount of information inherent to the transition from one timestep to another. It is impossible to represent the transition from one timestep to another in less than $O(n)$ memory without recomputing values. Thus, the lower bound for the worst-case scenario is $O(n * t)$ memory. For simulations where storing the position of every particle at every timestep is infeasible, the applicability of our algorithm is limited. In such a scenario, a rolling window of timesteps, or a different data structure altogether, may be more appropriate.

5 FUTURE WORK

Our work can be expanded in several ways, both technically and theoretically. Our technical implementation is not optimal, and can be made more robust. Python was used as our language of choice for ease of implementation, but using a compiled language such as C++ would result in better overall performance. A C++ implementation was attempted, but ultimately dropped in the interest of time due to the ease of a Python implementation.

On the theoretical side, our algorithm can be improved by handling intersecting segments. Given that segment intersections are relatively rare in our simulation, the best approach is to split one segment of each pair of intersecting segments at the intersection

point. Doing so would not incur a memory usage or running time penalty significant enough to affect the overall order of magnitude.

Our simulation design can also be improved in the future. Currently the simulation doesn't not guarantee that the path the particles take don't intersect in one timestep. This could remedied with an event-based simulation, where the time until the next event is calculated. Using such a simulation, it could be guaranteed that particles paths will never cross, as the particles will either interact or collide.

In terms of simulation speed, there could be methods to increase the simulation speed. Currently each particle is kept in a Map data structure in python. At each timestep the particles position and velocity is altered. For the Wierner process simulation, the simulation could be done even more efficiently by storing all particles in a $2xn$ numpy matrix. Matrix operations are heavily optimized and progressing the simulation by one timestep would involve a matrix addition. This could give a big performance boost.

The simulation could also be further expanded upon to include other types of particle-particle interactions. In many simulations particles have to modeled to include many different interactions. Currently the simulation will only generate data from elastic collisions between particles.

6 WORK DIVISION AND TIME SPENT

Theodore Wu created the simulation engine, and performed all associated work. William Allen implemented the segment tree described herein, and researched existing segment tree implementations. The proposal, proposal refinement efforts, presentation, and final report were a joint effort.

Theodore spent most of his time on the elastic collisions simulation. Unlike the other two simulations more work was needed to ensure that each particles position was resolved correctly. There is a surprisingly sparse amount of material about creating particle simulations. There are many software packages that can handle these types of simulations, but the material about creating your own from scratch is not easily accessible. Furthermore these free simulations, are often much more specialized than what we needed for the purposes of this project. Much of time on the simulation was spent research ways to properly implement a correct simulation.

William spent a quantity of time working with CGAL's segment tree implementation over the course of the first 3-4 weeks of the project. After discovering that CGAL's segment tree implementation is broken, and cannot be easily patched, it took about a day and a half of work to implement and debug a basic homemade segment tree implementation. Creating the required presentation and writing this report took about two days of work, in addition to miscellaneous amounts of time over the course of the preceding weeks to do basic setup tasks.

ACKNOWLEDGMENTS

Barbara Cutler, for her excellent Computational Geometry class, and the segment tree slides she put together.

REFERENCES

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. 2008. *Computational Geometry* (3rd ed.). Springer, Chapter 10.

[2] Nabil Ibtehaz, M. Kaykobad, and M. Sohel Rahman. 2021. Multidimensional segment trees can do range updates in poly-logarithmic time. *Theoretical Computer Science* 854 (2021), 30–43. https://doi.org/10.1016/j.tcs.2020.11.034

[3] Steven Lalley. [n. d.]. LECTURE 5: BROWNIAN MOTION. http://galton.uchicago.edu/~lalley/Courses/390/Lecture5.pdf

[4] Vobarian Software. [n. d.]. 2-Dimensional Elastic Collisions without Trigonometry. https://www.vobarian.com/collisions/2dcollisions2.pdf