5-2023

Jonathan's Rust Adventures and the Quest for the Numerically Stable Soft-Sphere Integrator

Jonathan Pascal Rotter
Trinity University, jonathan.p.rotter@gmail.com

Jonathan's Rust Adventures and the Quest for the Numerically Stable Soft-Sphere Integrator

Jonathan Pascal Rotter

Abstract

In this thesis I will implement a numerically stable soft-sphere collision model using Hooke's Law as the restoring force. This model allows choosing a desired coefficient of restitution and maximum penetration depth in order to generate appropriate values for the spring constant and damping. Then I will explore the applicability of various languages to the field of kD-tree based N-body simulations, concluding that Rust is competitive in both the time and memory usage to C/C++, which are the languages traditionally used for high-performance applications. Additionally, I found that the usage of higher-level languages like Java come at significant time or memory usage costs, making them of little value in the field of large astrophysics simulations. Lastly, as soft-sphere simulations require much smaller time steps for integration compared to hard-spheres, I implemented and benchmarked a priority-queue based adaptive time step system to only take the small time steps when needed. The benchmarks showed a significant speedup compared soft-sphere simulations without this adaptive time step method at roughly 8x faster. In direct particle-particle testing, the adaptive time step showed that with as little as 15 steps during a collision, which the adaptive time step method allows me to set, stable collision handling can be achieved across a spectrum of impact speeds and particle size ratios.

Acknowledgments

I'd like to thank my advisor Dr. Lewis for his major contribution to and guidance in this endeavour. I'd also like to thank the Trinity University department of Computer Science for funding the trip to the 2022 International Conference on Computational Science and Computational Intelligence to present the paper on kD-tree performance in various languages. Additionally I'd like to thank the other committee members, Dr. Fogarty and Dr. Mehta, for their insightful feedback.

Jonathan's Rust Adventures and the Quest for the Numerically Stable Soft-Sphere Integrator

Jonathan Pascal Rotter

A departmental senior thesis submitted to the
Department of Computer Science at Trinity University
in partial fulfillment of the requirements for graduation
with departmental honors.

April 14, 2023

Thesis Advisor

Department Chair

Associate Vice President
for
Academic Affairs

Jonathan's Rust Adventures and the Quest for the Numerically Stable Soft-Sphere Integrator

Jonathan Pascal Rotter

Contents

List of Tables

List of Figures

Chapter 1

Background and Introduction

N-body simulations have been part of the field of planetary ring dynamics since the late 1980s [3, 27]. Early models of planetary rings used descriptions from thermodynamics and statistical mechanics. However, the Voyager showed that there is structure in nearly all visible scales in Saturn's rings, which highlighted that equilibrium models cannot accurately describe the full dynamics of Saturn's rings. Since computers are limited in computational power, particle simulations as early as the numerical models by Wisdom and Tremaine [27] make use of periodic "sliding-brick" boundary conditions, so that one area in the rings can be simulated accurately without having to simulate the entire ring. Their method has become the standard approach for local cells. Their approach also modeled the particles as hard-spheres that bounced off each other in discrete events that were processed sequentially.

This hard-sphere model with discrete collisions has been used by many following researchers in other simulation codes [23, 15]. Another method, the soft-sphere model, has also seen extensive use by Salo [25]. Instead of discrete events, particles are allowed to overlap and are pushed apart by a restoring force. The benefit of the soft-sphere model is that it allows exploring high density ring systems as particles stacked and rolling over

each other isn't feasible in hard-sphere codes as the number of collisions to process becomes simply too large. Thus this model also finds use also in granular flow as there the particles are often densely packed. The restoring force is modeled as a Hook's Law spring force, so that the restoring force is proportional to the overlap distance. As rubble-pile asteroids are also made of packed spheres, this soft-sphere model has made its way into the PKDGRAV package [26]. See [24] for a review of the basic math behind both hard- and soft-sphere collisions in planetary rings.

In this thesis, I will describe my efforts making stable soft-sphere simulations. Additionally, this work is part of an modernization effort to re-implement Dr. Lewis' existing C++ ring simulations in Rust [10]. The reason why Rust was picked for these simulations is that it makes use of an ownership model to guarantee both memory and thread safety at compile time without a garbage collector [16]. Thus, it should facilitate making more robust code than C++ but without the performance penalty incurred by many higher-level languages like Java that make use of a garbage collector or reference counting.

The work is divided roughly into three projects. First, I will attempt to find a derivation for the spring and damping constant in Hooke's Law that result in numerically stable soft-sphere collisions. Secondly, as part of the move from C++ to Rust, I wanted to ensure that Rust is competitive in both memory and time usage compared to C++ as a penalty in either department would lower the maximum simulation size and duration that is feasible compared to C++. Lastly, I attempt to scale up the testing simulation into a full ring simulation with sliding brick boundary conditions and the Hills Force. As only the particles in one cell in the rings are simulated, the central body isn't actually a particle in the simulation. Rather, the simulation cell, i.e. the reference frame is rotating around the central body. To make the particles behave like they're orbiting a central mass, the Hills Force is added which is a linearized solution to the gravity the particles would feel from the

central mass in such a rotating reference frame like this simulation setup. I also attempt to address one of the major downsides of a soft-sphere simulation, and that is that a much smaller time step is required in order to keep the simulation numerically stable. A collision between two particles happens in a dozen or more steps in the soft-sphere model, and if a collision occurs only over a few time steps, the integration of the forces will be flawed resulting in nonphysical things such as a drastic increase of kinetic energy, leading to particles "blowing up" in the simulation. As such, I introduce a priority queue to process nearby particles in small, adaptive time steps, so that faster moving particles can get the small time steps they need without slowing down the whole simulation.

Chapter 2

Soft Sphere Collisions

In this chapter I attempt to derive an approach to soft-sphere collisions that allows choosing a desired coefficient of restitution (the ratio of exit vs entry velocity), and ensures that particles do not overlap too much, as that would be nonphysical. I will also compare the derivations to those of Schwartz et. al. [26], which looked at soft-sphere granular flow. This work was originally published in [7].

2.1 Approach

Various parameters need to be set when modeling soft-sphere collisions. The spring force which I use to simulate the interactions between two colliding bodies are governed by k, the spring constant, and c, the damping constant for kinetic friction. These are chosen such that the bodies do not overlap by more than 10% as that would be non-physical since the overlapping in the simulation represents deformation. I represent the overlap using δ. The other constraint is the coefficient of restitution, ϵ, such that $v_f = \epsilon v_i$, where v_f is the relative speed after the collision and v_i before. In hard-sphere models, matching this

constraint is easy as collisions are discrete events and so the simulation has direct control of the velocity of the particles before and after. Often ϵ is even velocity dependent in hard-sphere simulations. In soft-sphere collisions, the coefficient of restitution is a product of the many time steps during which a collision occurs. In planetary ring simulations, experimentally derived coefficients for ultra-cold ices from [4] and [5] are most commonly used. Though recreating these experimental values is preferred, in practice soft-sphere simulations ause a constant value, $\epsilon \approx 0.5$.

Since the collision is driven by k and c, I need to derive values that would produce the desired values for ϵ and δ. Since we treat the restoring force as a damped harmonic oscillator, the motion over half a cycle is simply

$$x = -e^{-\gamma t} A \sin \omega_l t \tag{2.1}$$

where x is the overlap between the particles, i.e. the distance between their centers minus the sum of the two radii.

$$\gamma = \frac{c}{2m}$$
$$\omega_l = \sqrt{\omega_0^2 - \gamma^2}$$
$$\omega_0 = \sqrt{\frac{k}{m}}$$

From here, there are multiple derivations of k and c depending on which assumptions were made.

Let δr be the penetration depth, which models the deformation of the soft spheres. We want to aim for δr to be 2% of the radius, and it should definitely be less than 10% of the

radius as such a large deformation would be physically unrealistic.

Let v_i be the impact speed, v_f be the exit speed and the coefficient of restitution be

$$\epsilon = \frac{v_f}{v_i} \tag{2.2}$$

2.1.1 Rotter

This is my derivation for k and c and it goes as follows. To derive k and c, I must find the first minimum of equation 2.1 which can be computed by solving for when $\dot{x} = 0$. This occurs at $\omega_l t = \arctan\left(\frac{\omega_l}{\gamma}\right)$.

The assumption I make is that $\omega_l >> \gamma$ and so $\frac{\omega_l}{\gamma}$ is large and $\arctan\left(\frac{\omega_l}{\gamma}\right) \approx \frac{\pi}{2}$. In other words, I assume that the introduction of damping does not significantly move the t value of the minimums and maximums and so $t \approx \frac{\pi}{2\omega_l}$. My x value is the penetration depth δr so $\delta r = -Ae^{-\frac{\gamma\pi}{2\omega_l}}$ at the minimum.

To find A, I take the first derivation at $t = 0$, the time of impact, to get $v_i = v(0) = -A\omega_l$ so $A = \frac{-v_i}{\omega_l}$. I can plug this into the equation for the penetration depth to get

$$\delta r = \frac{v_i}{\omega_l} e^{-\frac{\gamma\pi}{2\omega_l}} \tag{2.3}$$

The desired penetration depth is a known number, 2% of the radius, and I can estimate the impact velocity and so only ω_l and γ are unknowns here.

To obtain a second equation so that I can solve for the two unknowns, I look to the coefficient of restitution, ϵ. The energy of a damped harmonic oscillator is $E(t) = \frac{1}{2}kA^2e^{-2\gamma t}$, and if I plug in $t = 0$ and $t = \frac{\pi}{\omega_l}$ for the beginning and end of the collision, respectively, I find that the ratio of E_{initial} and E_{final} is $e^{-\frac{2\gamma\pi}{\omega_l}}$. $E = K + U_{\text{spring}}$, but at $t = 0$ and $t = \frac{\pi}{\omega_l}$, $x = 0$ and so $U_{\text{spring}} = 0$. Thus the energy ratio is also the ratio of kinetic energy.

$\frac{K_f}{K_i} = \frac{\frac{1}{2}mv_f^2}{\frac{1}{2}mv_i^2} = \frac{v_f^2}{v_i^2}$.

$$\frac{v_f}{v_i} = \epsilon = e^{-\frac{\gamma\pi}{\omega_l}} \tag{2.4}$$

Combining equations 2.3 and 2.4, I can solve for γ and ω_l, and from there k and c can be determined as $\gamma = \frac{c}{2m}$ and $\omega_0^2 - \gamma^2 = \omega_l^2$ and $\omega_0^2 = \frac{k}{m}$.

$$k = m\left(\frac{v_i}{\delta r}\right)^2 \frac{\epsilon\left((\ln\epsilon)^2 + \pi^2\right)}{\pi^2} \tag{2.5}$$

$$c = 2\ln\epsilon\sqrt{\frac{k\mu}{\pi^2 + (\ln\epsilon)^2}} \tag{2.6}$$

Here, μ is the reduced mass of the two particles, ϵ the desired coefficient of restitution and δr the desired penetration depth. The derived equation for c agrees with the one presented by Schwartz et al. for drag in the normal direction, but the k I determined is slightly different.

2.1.2 Lewis

This is the derivation for k and c of Dr. Lewis. Like mine, it assumes that $\omega_l >> \gamma$ and so the first minimum is at $t \approx \frac{\pi}{2\omega_l}$. However, it adds the approximation that $\omega_l \approx \omega_0$, which is true if $\gamma << \omega_0$. So $A = \frac{-v_i}{\omega_0}$ instead of $\frac{-v_i}{\omega_l}$ and the time at which the collision ends is $t_f = \frac{\pi}{\omega_0}$ and the maximum penetration is at time $t_{max} = \frac{\pi}{2\omega_0}$. The derivations from 2.1.1 can mostly be reused with a few replacements of ω_l with ω_0, resulting in

$$k = m\left(\frac{v_i}{\delta r}\right)^2 \tag{2.7}$$

$$c = \frac{2\mu\omega_0 \ln\epsilon}{\pi} = 2\ln\epsilon\frac{\sqrt{k\mu}}{\pi} \tag{2.8}$$

These equations are mostly simplified versions of equations 2.5 and 2.6 and interestingly, equations 2.5 and 2.7 only vary by a constant factor of ≈ 0.52 (for $\epsilon = 0.5$). The $\ln(\epsilon)^2$ term missing from this derivation of c is approximately 0.48 (for $\epsilon = 0.5$).

2.1.3 Schwartz

Schwartz et al. presented their own derivations in their paper, which I will summarize here since all three of these derivations are being compared to each other. Schartz et al. also considered glancing collisions, but for this we only consider head on collisions, i.e. movement along the normal axis of the spheres (Equations 4 & 15) and tangental forces will be ignored. The notation is adjusted to make it consistent with the previously setup notation. Thus the equations are

$$k = m\left(\frac{v_{imax}}{\delta_r}\right)^2 \tag{2.9}$$

$$c = 2\ln\epsilon\sqrt{\frac{k\mu}{\pi^2 + (\ln\epsilon)^2}} \tag{2.10}$$

where v_{imax} is the maximum expected impact velocity, rather than the impact velocity of the current collision. As previously mentioned, their equation for c matches my derivation and their equation for k is similar to Dr. Lewis'.

2.2 Numerical Approach

Since during a collision a particle's velocity drastically changes in a few time steps, keeping these collisions numerically stable is a challenge. While gravitational forces usually change very gradually, $1/x^2$ for large x, the spring force is linear with respect to the position. If the integrator cannot resolve a collision well, collisions become nonphysical and particles leave with much higher energies than they came with. The most straightforward solution to the problem is to use smaller time steps, however, this increases computation time with wall clock time being inversely proportional to the size of a time step. Another approach is to use a higher-order integrator, but these often have to store partial steps in memory and/or do multiple passes over the system to calculate intermediate forces. As large astronomical simulations are often constrained in memory and time, the overhead of higher-order integrators make them sub-optimal and so I want to find a numerically stable approach with a maximum time step and minimal memory overhead.

Two integrators were implemented for numerical testing, a 2nd-order leap frog as described in [26] and a 4th-order integrator that makes use of jerk, the derivative of acceleration, in addition to acceleration [6]. As acceleration changes quickly during a collision, jerk should add context on how acceleration acts around the current time step. A common problem in soft sphere collisions is if two particles move too much within one time step, effectively going from barely touching to having significant overlap, the resulting spring force will explosively separate the particles and the chances of properly resolving this collision are slim. Higher order integrators like Runga-Kutta could help with better resolving collisions, but they require significant memory overhead to store intermediate steps and the integrators also require calculating forces in intermediate steps, and as such don't mesh well with the memory constraints of large simulations, making them a bad candidate for

planetary ring simulations. The modified leapfrog integrator that makes use of jerk[6] does have some memory overhead for handling jerk and correcting for it, but it does not add additional particle traversals.

Since the force is a piecewise function, i.e. gravitational force when the particles are apart and the linear restoring spring force when overlapping, I experimented with a smoothing function to transition the force between the two sides. As the piecewise nature creates a discontinuity in the force and thus acceleration, the smoothing function might help the integrator better resolve the transition. Additionally, the jerk as a derivative of acceleration can't be influenced by points beyond the discontinuity, which smoothing should also assist with. To test the hypothesis that smoothing leads to better collision resolution, tests were run with both the smoothing function and also without, leaving the discontinuity in place. As a smoothing function, a sigmoid was used with various widths tested.

The test is setup by making two particles close to each other (usually that the initial separation is a tenth of the sum of the radius) and the velocity is set such that they are heading towards each other. The parameters are set to resemble common planetary ring simulations as the end goal is the integration of soft sphere into these simulations.

As such, the unit of mass is the mass of the central body (usually Saturn), the unit length is the semimajor axis of the orbit (usually 100,000km), and the unit time is one orbit divided by 2π. This unit system results in G, Newton's gravitational constant, being one.

The parameters varied and tested are:

- time step: $0.006, 0.003, 0.001, 0.0003, 0.0001$
- desired impact speed: `1e-8, 1e-7, 3e-7, 1e-6, 3e-6`
- sigmoid width modifier: $0.1, 0.03, 0.01$

- radii: `1e-7, 3e-8, 1e-8`
- all three k & c derivations
- the two integrators
- with the sigmoid smoothing function and without

The impact speed being $\frac{d}{dt}|\vec{x_i} - \vec{x_j}|$ where $\vec{x_i}, \vec{x_j}$ are the positions of the particles in the simulation.

The radii of the second particle was tested with all sizes of radii that are the same or smaller than the first particle's radii. The sigmoid width modifier is used to adjust how far the smoothing function reaches (i.e. the size of the transition area between gravity to restoring spring force) using the following formula:

$$\sigma\left(\frac{4}{\text{width modifier} \times \min(\text{radii})}\delta r\right)$$

where σ is the sigmoid function:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

To prevent floating point errors from exponentiation of large numbers, if $|x| \geq 100$, the implementation of the sigmoid function returns the approached value, i.e. zero for large negative numbers and one for large positive numbers.

The testing setup is along the x-axis, with the two particles being placed at $x = -1.0000005 \times r_0$ and $x = 1.0000005 \times r_1$, respectively. As such, they are not colliding at the beginning but very close to it and the collisions are effectively 1D. This is done to maximize the control over the velocity at impact as gravity will speed them up as they approach each other. The setup is symmetrical with respect to velocity, so each particle's

velocity is half of the desired impact velocity and they are headed toward each other such that the relative velocity between them is the impact velocity.

To gather test results, the position of the particle is recorded at every time step. The first time step when they overlap, i.e. the distance between them is less than the sum of their radii, the relative velocity is recorded as the velocity at impact. This will be slightly different than the starting velocity due to gravity and the effects of the smoothing function. Then, once a time step is reached at which the particles are no longer overlapping, the relative velocity is recorded again, this time as the exit velocity. The test is concluded after one collision completes between the particles. Another measurement is the maximum penetration depth, which is checked every time step by finding the largest overlap seen during the collision. The coefficient of restitution is computed from the impact and exit velocities, and the maximum penetration depth percentage is computed per particle as the particles may not have the same radii. The number of steps the collision took is also recorded.

Since there are so many variables, tests are automatically checked for non-physical results and are marked as successful or failed depending on the checks. One check is that the particles may not pass through each other, i.e. the centers of gravity may not pass each other. Other checks consist of testing for if the particles are stuck to each other as damping does slow them down during the collision, testing if they move apart before colliding, testing if the velocity is non-finite, and testing if they did not complete the collision in the given time frame.

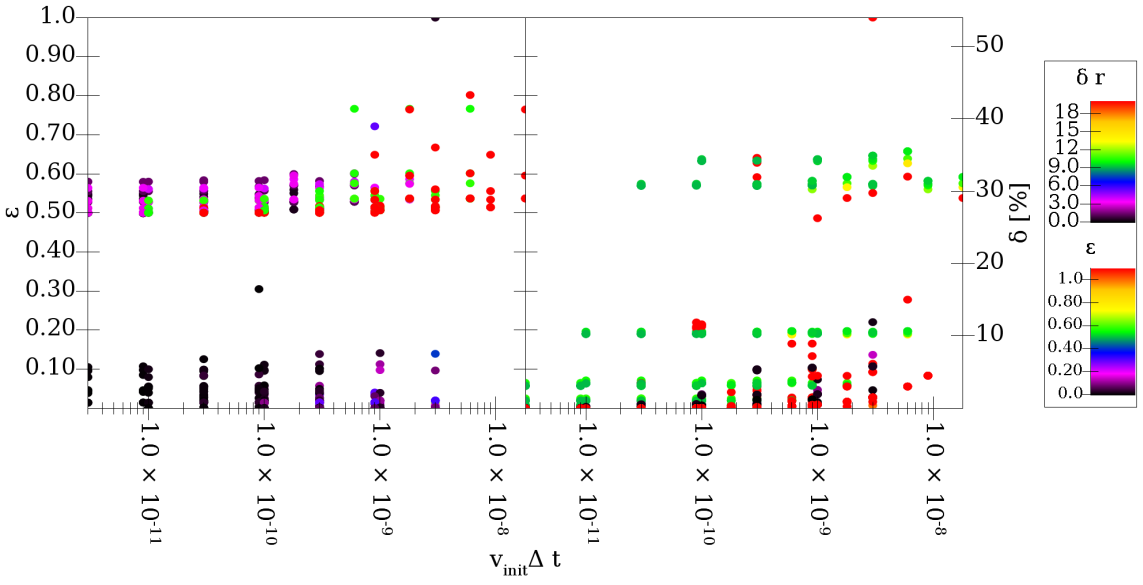

Figure 2.1: Plot showing the values of ϵ and δr vs. distanced traveled during the first time step of a run. The left side shows ϵ for the data set while the right shows δr for the data set. The colors on the left side show δr while the colors on the right show ϵ.

2.3 Results

A summary of the almost 5000 simulations run can be seen in figure 2.1. The x-axis is the distance covered in the first time step, which is the initial velocity times the time step. This is significant as the more distance covered by a step, the more challenging resolving the collision is going to be. On the left side the particles are plotted against the measured values of ϵ, the coefficient of restitution, and on the right they are plotted against the penetration depth percentage. For each side, the color indicates the value of the other variable. The gradient for both are set-up in such a way that green, blue, and magenta are mostly acceptable while yellow to red are not. Since the range of the radii of the particles are 10^{-8} to 10^{-7}, simulations where the distance covered in one time step is greater than

10^{-9} have a very low chance of producing the desired penetration depth as with such large step sizes, a single step is a significant percentage of the radius of the particle.

Simulations that failed the checks for non-physical behavior are not shown on this plot. Additionally, only runs using the 4th order integrator and the Rotter derivation for k and c are shown here. One point of concern in the data shown is that even on the left side, where steps are small and so the simulations should be resolvable by the integrator, there are some red data points, representing misbehaving collisions.

Another note on the plot is that many collisions have extremely low coefficients of restitution, indicating that many particles are sticking or almost sticking together after a collision. This is partly expected in ring simulations as some particles will end up resting on each other, but since the aim is to have controlled collisions with an $\epsilon = 0.5$ when calculating k and c, this does represent an area of further research on why this is happening.

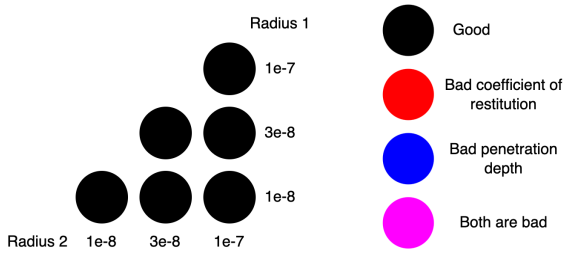

Figure 2.2: Key for figure 2.3 and 2.4

Figure 2.3 and figure 2.4 is another summary of the results, this time in a way to see which settings worked and which didn't since this work is focused on identifying methods that are numerically stable. The left panel represents the 2nd-order integrator and the right panel is for the 4th order integrator in both plots. Both axes are $\log_{10}$ scale due to the test parameters scaling in orders of magnitude. The cluster of dots represents all the combinations of the radii we tested, $10^{-7}, 3 \cdot 10^{-8}, 10^{-8}$. Note that each cluster used the

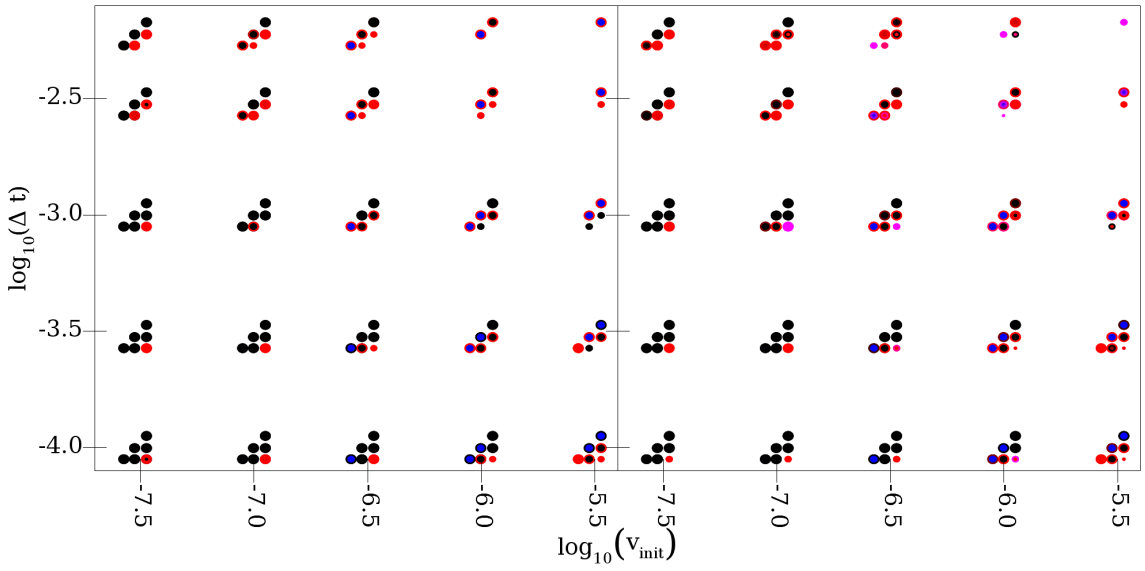

Figure 2.3: This plot shows the results of simulations in a way that enables comparing the different approaches to calculating k and c. Each "triangle" of dots shows variations in particle size while different sized dots were used for the method of calculating k and c. See figure 2.2 for details on the triangle and colors. Each dot is three differently sized dots on top of each other, the biggest being Schwartz et. al. derivation, the middle one the Lewis derivation and the smallest dot is the Rotter derivation.

same initial velocity and same time step. In one cluster, the bottom left corner is for the smallest radii, 10^{-8} and 10^{-8}, and the top right corner is for largest radii, 10^{-7} and 10^{-7}. The diagonal thus is where the particles have the same size, and the bottom right three dots are the combinations of different radii sizes, with the very bottom right corner being the collision of 10^{-7} and 10^{-8}.

The coloration indicates whether the measured ϵ and δr are acceptable. Red indicates that $\epsilon > 0.6$, and blue that $\delta r > 10\%$ of the particle's radii. If both are reasonable, the dot is black and if both are unacceptable, the dot is magenta. For figure 2.3, each dot is also composed of three superimposed circles of increasing size, each indicating one k and

c derivation. Schwartz et al is the largest, the Lewis derivation the middle one, and the Rotter derivation is the smallest. If one derivation succeeded while another didn't, a donut can be seen. However, most of the time all methods succeed or fail which corresponds to the fact that the derivations are quite similar as discussed in 2.1. In the case of figure 2.4, the small circle represents no blending and the large circle blending with a sigmoid function.

As seen in the two plots, figures 2.3 and 2.4, the bottom right corner representing the mismatched particle radii simulations are the most difficult to resolve. The bottom right corner is missing the most in the plot and is often also marked red. Missing data points means that those simulations were not numerically stable and failed the aforementioned automatic checks. This is an issue for applicability to numerical ring simulations as objects often have a large size distribution. For the hopper experiments, particles are uniform or roughly uniform [26]. Some ring simulations also feature moonlets, which means the integrator has to contend with differences in radii on the order of 100x or 1000x between background particles and moonlets. The instability given a 10x difference of radius seen in these results suggest that more work is needed before soft sphere collisions can be used in large scale simulations, especially those with moonlets.

For figure 2.4, the most noteworthy result is that the addition of a blending function doesn't change the output. Scaling the width of the blending function also didn't have a big effect. As the simulations succeeded or failed the same with and without the blending function, the use of a blending function seems to not be worth it as it introduces complexity to the simulations.

Another note is that when aiming for various values for ϵ, the numerically measured value for the coefficient of restitution does roughly match the ϵ value put into the code. Plots like figure 2.1 stay mostly the same except for the shift in ϵ values measured. However, lower values of ϵ do seem to cause less particles to stick together, which is rather non-intuitive

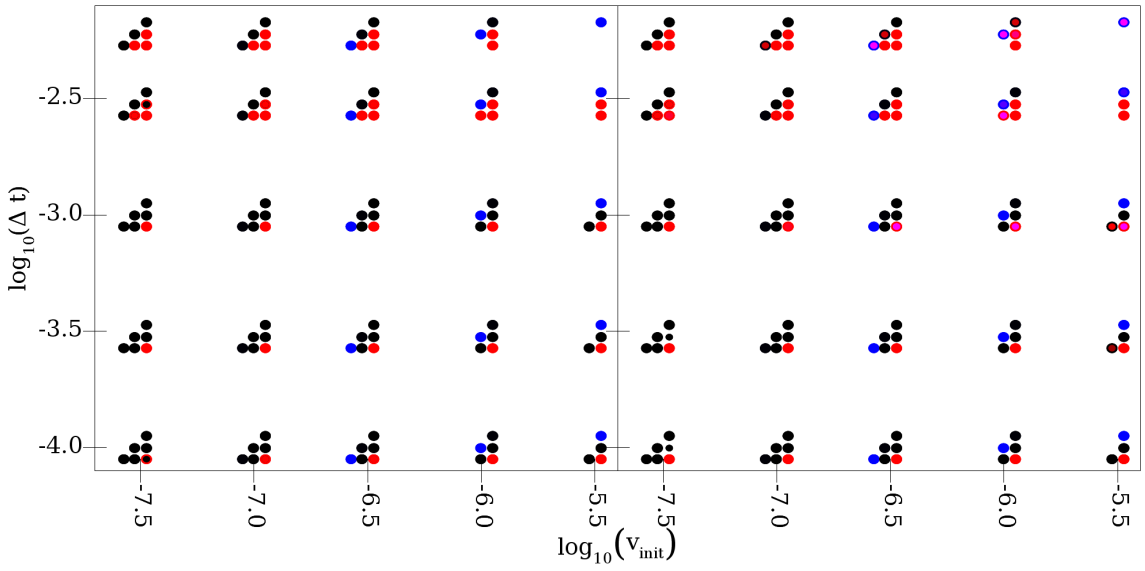

Figure 2.4: This plot shows simulations results in a way that enables comparing the usefulness of a blend functions. Each "triangle" of dots shows variations in particle size while different sized dots were used for blending with a sigmoid (small) and no blending (large). See figure 2.2 for details on the triangle and colors.

and is a point worth researching in the future.

2.4 Summary

The two key conclusions from this exploration of numerical stability of soft sphere collisions are one, the assumptions used when calculating k and c do make a difference. The two derivations presented here show a better measured coefficient of restitution compared to the derivation presented in [26]. Secondly, the use of a smoothing function between gravitational and collision forces and the jerk-based 4th order integrator did not improve the results as desired. Comparing the left and right frames in figures 2.3 and 2.4 shows little improvement at all. As such, the extra complexity of the smoothing function and memory overhead of

computing and tracking jerk for the 4th order integrator are not worth it.

Another point is that the bottom row in figures 2.3 and 2.4, the one with the smallest time step, was hoped to be all black circles representing good results but that did not happen. Smaller time steps are overall better, but increase simulation times drastically and the increase of accuracy doesn't match the increase of time cost.

The current hard-sphere collision code used by Dr. Lewis has time steps of $2\pi/1000$, which is roughly the largest time step analyzed here, meaning that a switch to soft sphere will have to come with a 10x or 100x decrease of time step used and as such a 10x or 100x increase of run time. The hard-sphere code currently uses a priority queue to handle collisions, which I will try to translate to soft-sphere as gravity calculations are expensive and change a lot slower than collision forces.

Another major challenge that needs to be resolved for soft sphere forcing to be viable in planetary ring simulations is the accurate resolution of collisions between particles with vastly different radii, up to 1000x. Many of the simulations in this analysis did poorly with only a 10x difference in radii. Other methods have used variable time step integrators to go around this issue [25], but this introduces issues with parallelizing the code, which is needed for processing simulations on clusters.

Chapter 3

N-Body Performance with a kD-Tree: Comparing Rust to Other Languages

3.1 Introduction

With derivations to resolve soft sphere collisions in hand, it is now time to write a soft sphere collision code base. However, this work also includes modernization of the existing C++ ring simulation codes. As simulations can take weeks or months to run, performance is critical. Thus I will implement a kD-tree N-body benchmark in C, C++, Rust, and a few other languages as reference. Specifically, Go, Java, Typescript on Node.js and Python will also be included in the benchmarks. Beside being a nice comparison to other languages, this shows whether the effort of writing in a systems language like C/C++ and Rust is worth it and whether Rust is competitive with C++ for the purpose of ring simulations.

This work was originally published in [8].

As mentioned above, for this benchmark I will use a kD-tree, which is a binary search tree where every internal node represents a plane splitting the space, partitioning space. The particles on one side of the plane will be in the left child, while those on the other side of the plane will be in the other child. As such, nearby particles will be nearby in the tree as well. Since gravitational forces are between every pair of particles, computing it would be an $O(n^2)$ problem. However, the approximation taken here is that if there is a cluster of particles far away, such that the angle made between rays going to either end of the cluster, called the opening angle, is some small value, the cluster is approximated as a single particle. This is where the kD-tree enters the picture as when the gravitational forces are computed for one particle, for each node, if the opening angle is sufficiently small, I approximate all children of the node, i.e. all particles in the area of space represented by that node, as a single object. This makes computing the gravitational forces roughly $O(n \log n)$, a vast improvement that allows running much larger simulations in a reasonable amount of time.

There are benchmarks available for N-body without the kD-tree, which is $O(n^2)$ [1]. However, since the addition of the kD-tree significantly changes how the algorithm works and involves a lot of tree traversal instead of simple iteration comparing particle-particle pairs, the existing N-body benchmarks are not as relevant to the kD-tree simulations as it might appear on the surface.

3.2 Approach

The simulation consists of a disk of small particles around a central body, uniformly distributed in distance from the central body. A picture of this together with the kD-tree's

partitioning of the space can be seen in figure 3.1. The green dots, almost too small to see, are the particles and the black lines represent where the kD-tree divides the space, until there are only at most 7 particles in a leaf. All implementations and details on parameters can be found at [12].

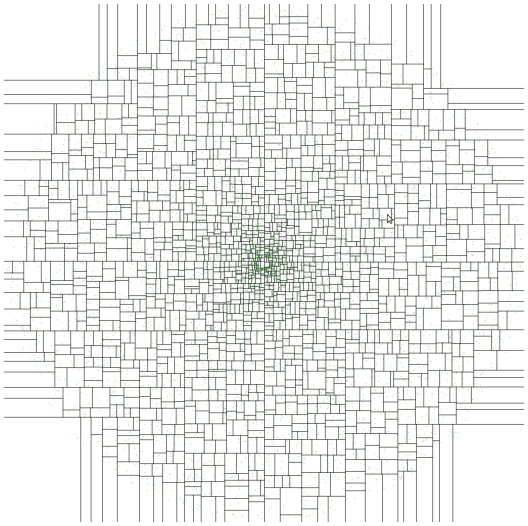

Figure 3.1: A simulation with 10,000 particles and the kD-tree partitioning of the space after 90 time steps.

3.2.1 Language Selection

Since Dr. Lewis' simulation code is written in C++, and we are looking at using Rust, these two languages have to be benchmarked. Additionally, we also look at C, as another system level language. Besides these, common languages from the RedMonk programming language rating were chosen [17]. Only one language per platform was chosen, e.g. between Java, Kotlin, and Scala, only one language is implemented as a good implementation between these should translate into roughly the same performance (Through idiomatic code might have differing performance, this was seen by the Scala Center [2] using the methodology from

[18]. Languages like Clojure which are dynamically typed on a statically typed platform may also see different performance).

Though only one language per platform was tested, some variations in implementation in the same language were tested. Rust got an implementation with explicit SIMD and one without and Java an implementation in an object-oriented style and one using just arrays of doubles. The variations in Java are intended for comparing memory layout. Arrays of primitives are contiguous, while an array of objects only stores the pointer, so the actual objects are scattered throughout memory.

As some languages included have weak multi-threading support (Python & JavaScript), and the complexity of the kD-tree algorithm increases significantly when doing multi-threading, the implementations written are all single-threaded. Especially the building of the kD-tree does not easily lend itself easily to multi-threading, and to facilitate the ability to port the code to many languages, simplicity was chosen over multi-threading.

The languages tested are Rust, C, C++, Go, Java, Typescript, & Python.

3.2.2 You can write Fortran in any language

To make the code easily portable, it was written in a Fortran-style, using indexing and arrays, where the arrays live for the entire program in order to minimize-and for some languages completely eliminate-dynamic memory allocation. These features of arrays and indexing exist in almost every language, and so porting the code into a language mainly constitutes an adjustment of syntax.

Additionally, the original kD-tree implementation of Dr. Lewis's general simulation framework in C++ is written in this style, which is what we want to compare to [13, 14]. The general simulation framework is much more advanced than what is needed here, but it is a good sanity check for what performance I can expect from the C++ version. The Rust

implementation was written first and compared to the full, existing C++ version to ensure its correctness and reasonable performance. The Rust version then became the template for all other languages to keep implementations consistent. Even though there was a C++ codebase with a kD-tree already, for consistency the C++ version benchmarked was derived from the Rust version.

3.2.3 Benchmarking setup

The performance tests were run on a workstation with two Intel®Xeon®E5-2680 v3 CPUs and 64 GB of RAM. The Linux command `time` was used to standardize how timing data is collected and reported. Specifically, user time was used as the tool also reports system time.

For each language, a range of simulation sizes from 1000 to 1 million was tested, and each configuration was run five times and for 100 iterations.

3.3 Results

Table 3.1: Intel Xeon Timing Results (secs)

Language/ Style	**Number of Particle**			
	1000	***10,000***	***100,000***	***1,000,000***
Rust	0.57 ± 0.01	11.52 ± 0.05	198 ± 2	2960 ± 50
Rust SIMD	0.58 ± 0.03	12.6 ± 0.1	221 ± 2	3190 ± 50
C++	0.48 ± 0.02	10.07 ± 0.05	181 ± 6	3820 ± 30
C	0.50 ± 0.01	10.17 ± 0.02	176 ± 3	3770 ± 10
Go	0.91 ± 0.03	16.5 ± 0.3	262 ± 2	4830 ± 40
Java OO	1.92 ± 0.03	20.2 ± 0.7	350 ± 16	8570 ± 160
Java Array	1.7 ± 0.1	17.0 ± 0.3	290 ± 5	7520 ± 100
TypeScript	2.11 ± 0.05	40.3 ± 0.6	750 ± 30	22800 ± 700
Python	109 ± 3	1760 ± 15	27200 ± 300	–

For C & C++, gcc was used first. But given that Rust is built on LLVM, C & C++ were also tested using clang to see if gcc and clang would differ. Clang was a little faster, $< 10\%$, and so those are the results shown in table 3.1. Python was not tested at one million particles because its execution time would be on the order of a week, making it impractical to collect the requires five run times. Java was run using GraalVM 22.2.0 for Java 17.0.4, Typescript using Node v10.19.0, and Python using CPython 3.10.7. CPython 3.11 does come with performance improvements, but it was not yet released at the time this testing was being done.

In figure 3.2, the execution times can be seen as multiples of the standard (without explicit SIMD) Rust implementation for that simulation size. In this figure, Python has been left out since it is so slow that the performance differences between the other languages wouldn't be easily visible. See figure 3.4 for the plot with Python included. Figure 3.2, shows expected things, such as the fact that C/C++ and Rust outperform all other languages. Both Go and the JVM outperform Node.js. The JVM's startup time's cost can be seen as it performs its worst compared to Rust in the smallest run, while becoming much more competitive in larger simulation sizes as the startup cost becomes amortized over the runtime of the program. Go seems to have an edge on the JVM, as though in the middle-sized simulations they perform similarly, in the one million case Go scales better than the JVM. One thing that may be surprising to some is that even though both Python and JavaScript (what the TypeScript transpiles into) are dynamically typed interpreted languages, Python is much slower than JavaScript on Node.js.

3.3.1 Memory Usage

Memory usage is critical to the performance of a program. Fetching data from RAM takes ˜100 clock cycles. A fetch from cache, on the other hand, can be about 10x faster. Thus,

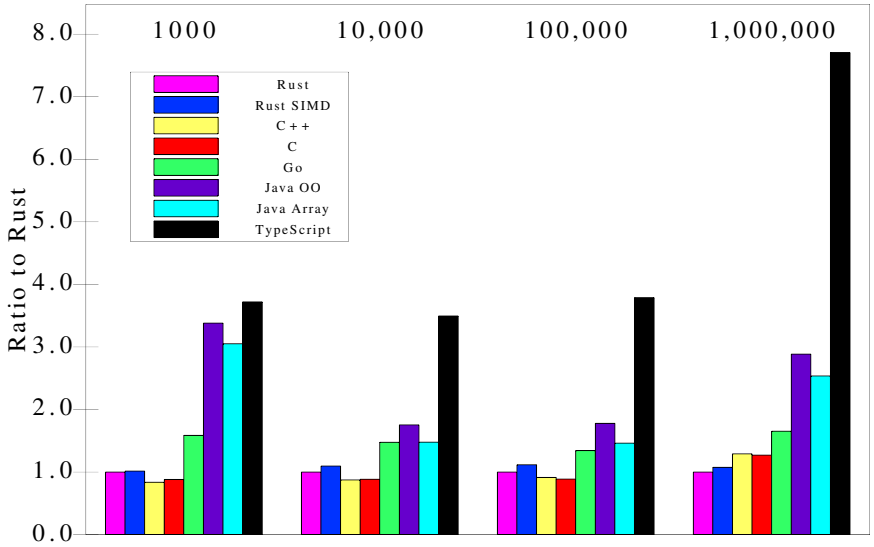

Figure 3.2: Execution times as a multiple of the standard Rust execution time for that simulation size, simulation size being the number of particles in the simulation.

for optimal performance, programs should use their memory in a cache-friendly way. The caching statistics of the processor used in these benchmarks, the Xeon®E5-2680 v3, are as follows:

L1 12 x 32 KB 8-way set associative instruction and data caches

L2 12 x 256 KB 8-way set associative caches

L3 30 MB 20-way set associative shared cache

Since this code is single-threaded, the distinction between shared vs. separate cache is unimportant.

The variation we tested within the JVM, Java OO vs arrays of primitives, is especially aimed at looking at how memory usage affects performance. To capture memory usage information, the GNU `time` command was used, specifying the `-v` flag[1]. The "Maximum

[1]Note that GNU `time` is distinct from the standard Linux `time` command/

resident set size" for each run can be seen in table 3.2. No implementation uses less than 1MB, so no simulation can fit completely into the L1 or L2 caches. However, for many languages like Rust, C, C++, and Go, the entire simulation for sizes less than one million fits into L3 cache.

Table 3.2: Resident Memory Usage in MB

Language/ Style	**Number of Particle**			
	1000	***10,000***	***100,000***	***1,000,000***
Rust	2.8	4.0	16	153
Rust SIMD	3.0	5.5	24	265
C++	3.8	5.7	19	202
C	2.3	3.2	15.5	153
Go	2.4	4.4	19.1	174
Java OO	163	154	198	531
Java Array	147	175	190	863
TypeScript	73	109	170	672
Python	12.5	10.9	101	–

From table 3.2, we can see Rust, C, C++, and Go have a much smaller memory footprint than the other languages. The JVM, perhaps unsurprisingly, consumes the most memory in most cases. The JVM is known for having a large memory footprint, and we clearly see that here. However, Node.js, especially for the larger simulations, is not much better than the JVM. They have much larger memory footprints because of the use of virtualized environments and memory models that don't allow easy control of where things are placed in memory. This can hurt caching. As an example, the C++ and Java OO implementation of a `Particle` look remarkably similar [12], featuring velocity and position in arrays of three doubles each, and a double for each mass and radius. However, these translate into very different memory layouts. For C++, all of this is turned into one chunk of memory big enough to fit the combined eight double-precision floating point numbers. In Java, however,

the object has two doubles and two references, plus some overhead. The references then point to array objects with three doubles, an integer length, and some overhead.

The system as a whole faces the same problem. A C++ `vector<Particle>` is a contiguous chunk of memory holding all the Particles, and so iterating through the vector is extremely cache friendly as the next element processed sits next to the current element in memory. In Java, an `ArrayList<Particle>` holds only references instead, and so the actual Particle data is not guaranteed to be close to each other as they may be anywhere in memory. At runtime, this both translates into memory overhead and having to follow references, adding memory fetches.

As an attempt to combat this, a version of the Java implementation was tested where instead of using an array of type `Particle`, multiple arrays were used storing the doubles directly (e.g. one array stored all the masses and one all the radii, etc.). Java treats arrays of primitives differently than normal objects, making the array and its values a single chunk of memory rather than using references. Thus the only objects we have are the arrays, of which there are four, one for position, one for velocity, one for mass, and one for the radii. This Java implementation consistently saw 10-20% faster results than the Java object-oriented implementation. However, this array-based implementation had the largest memory footprint, which may be due to the JVM's garbage collector copying objects to move them from short GC pools to long-term ones, and so (at least for a brief moment) the massive arrays would be in memory twice.

For Rust, it may seem strange that the SIMD implementation performs worse, but that is what the results consistently show. This may be due to increased memory usage, normally a 3D-vector is 24 bytes, 8 bytes per double-precision floating point value, but the SIMD used operates on four values at once and so the 3D-vectors carried along a superfluous double, increasing the memory usage per 3D-vector to 32 bytes. This increase

in size does give an explanation for why the memory usage is greater for the SIMD version for all simulation sizes. The superfluous double does get operated on, but if using SIMD instructions, that shouldn't make a difference as all four doubles should be operated on together. An independent test of explicit SIMD on the simpler O(n^2) N-body benchmarks does see increases in performance. In the case of these kD-tree benchmarks, it may be the case that the increase in memory usage offsets the gains by using SIMD.

3.3.2 Scaling

As mentioned earlier, a kD-tree based N-body simulation should scale roughly $O(n \log n)$. On the order of the particles tested in this analysis, one thousand to one million, a 10x increase should be around a 12-13x increase in execution time. The actual results can be seen in Figure 3.3, and are usually more in the ballpark of 18-20x. The JVM versions from 1k to 10k particles looks good, but that is because of the startup of the JVM skewing the 1k results to be much slower, which in turn makes the scaling look better. In reality, the startup cost of the JVM was just amortized over a greater execution time.

While scaling is relatively similar across the board (execpt for the previously mentioned JVM anomaly) for 1k to 100k, it is the scaling to one million particles where things get interesting. At this point, no simulation can fit in the L3 cache. (Rust, C, C++, and Go can all fit in the 30 MB L3 cache with 100k particles). We can see that Java and TypeScript don't scale well at this point anymore, but more interesting, C/C++ scales significantly worse here than Rust. It is currently not known why Rust performs so well at this point, but it will have to do with how memory is accessed. This good scaling makes Rust the fastest language at the one million mark. Python's behavior at this point couldn't be measured due to its slowness making it impractical to benchmark.

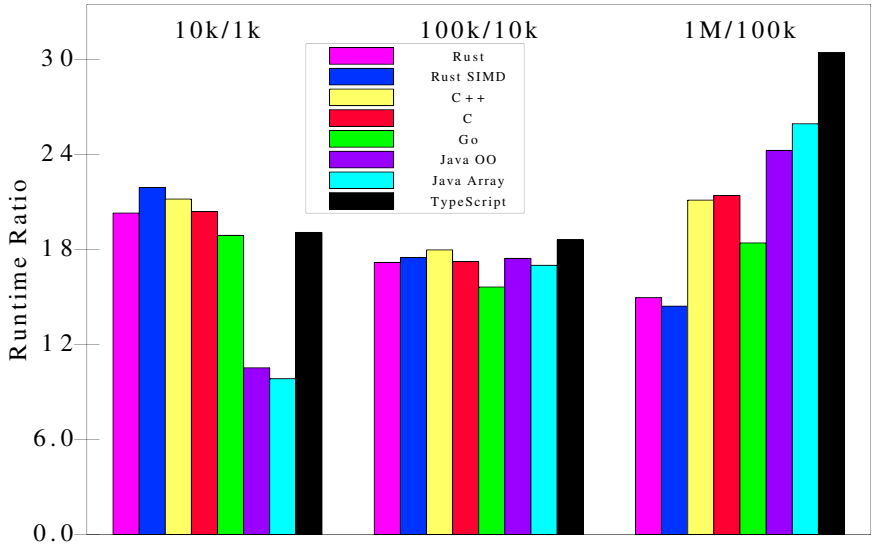

Figure 3.3: Execution time scaling with particle count. Each bar measures how well a language scales compared to itself, e.g. the 100k/10k for C++ shows that C++ took 18x longer for 100k particles than 10k particles.

3.3.3 Python is Horribly Slow

I've previously hinted at Python's slowness, but here it gets its own section because of how much it stands out. Figure 3.4 is the same as figure 3.2, but with Python added and the one million particle simulation removed. Now python does get better compared to other languages as the simulations grow, but it is still much worse than 100x the execution time of Rust. In fact, every simulation, including Node.js, finished the one million particle simulation faster than Python on the 100k simulation. This large difference in speed is also why it was impractical to test python for one million particles. Surprising as these results may be, they are in step with the Benchmark Game's n-body benchmark results [1].

The implementation whose results are shown is written in pure Python. I've made multiple attempts to use NumPy, a library for fast computation and array operations written in C, to speed up the Python code. I also tried the array-based strategy with NumPy that

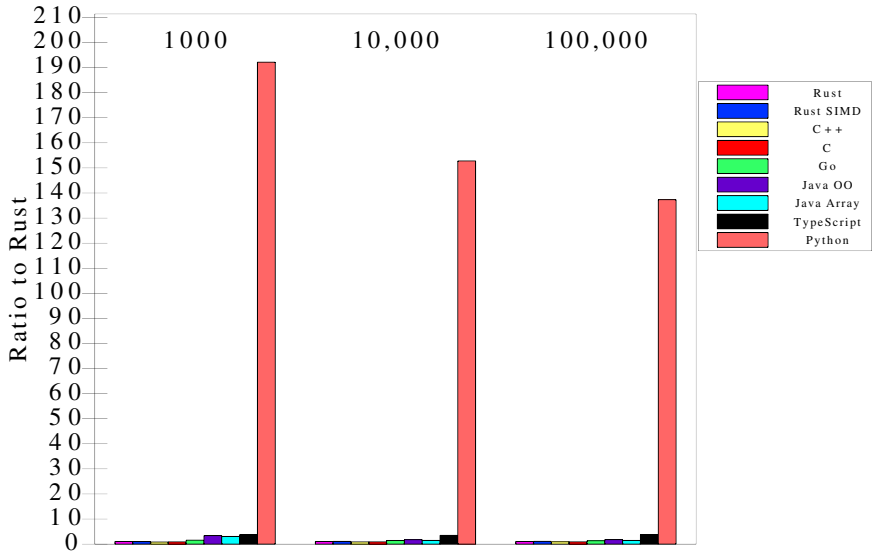

Figure 3.4: Execution times as a multiple of the Rust execution time for that simulation size including Python.

made Java slightly faster. However, all of these attempts resulted in around ˜2x worse performance than the pure Python implementation. The best Python implementations of n-body in the benchmark game also don't use libraries like NumPy [1], which suggests others have not found a way to speed this style problem up with NumPy either.

A note on NumPy is that it is designed to operate on arrays. However, the majority of the time, which makes the algorithm $O(n \log n)$, is spent walking the kD-tree, which has to happen within Python and not NumPy's code. NumPy can be used to integrate all the accelerations and velocities, but this is only $O(n)$, and so is out-weighted by computing gravitational forces by walking the kD-tree, which is $O(n \log n)$.

One possible speedup is Cython. It is a superset of Python which converts to C code that calls the Python API, and allows easy integration of C into Python. If the kD-tree was written in C or Cython, for example, it could then be called from Python code. However, then the code doing the heavy-lifting is not actually Python, undermining the point of a

benchmark.

Python 3.11, a particularly performance aimed update, released after this project was completed, and so was not used to benchmark this code. Python 3.11 claims 1.25x speedup over 3.10, but since Python lags behind by more than 100x, that speedup doesn't make a dent into the results of this project, and Python is still not a competitive language to do the heavy-lifting in a simulation.

3.3.4 PyPy to the Rescue?

While CPython is the reference implementation for Python, there is another prominent implementation. PyPy is an implementation using a JIT compiler that can reach 5x the speed of CPython, on average [21]. I've conducted some preliminary testing that showed speed-ups of 10-15x compared to CPython on this code. The codebase is mainly Pure-Python, which likely helps the JIT. In the traditional N-body problem, where every body is compared to every other body, PyPy records a 6.5x speedup compared to CPython3.7 [22]. From PyPy's benchmarks it also seems that the degree of the speedup is heavily dependent on the algorithm [21], so the finding of 10-15x speedup cannot be generalized to all Python programs. It rather seems that the kD-tree based code works particularly well with PyPy.

While PyPy is faster, it does introduce new issues. One is that at the time of this project, CPython was at version 3.10, while PyPy was at version 3.7 [20]. Additionally, PyPy's speedup originates not only from using a JIT, but also from semantic differences [19]. PyPy also uses much more memory than CPython, placing it close to the JVM in memory usage. It is also worth remembering that while PyPy is impressive in its speed, it is still 3x slower than Node.js and 10-20x slower than Rust.

3.3.5 Performance Where it Matters

While these details and performance at different simulation sizes may be interesting, in practice, the simulations that this benchmark tries to represent have at least a million particles. Thus the one million particle simulation is the most relevant. While the JVM might be doing okay in smaller simulations, the fact is that as the simulation size grows, it gets worse and worse compared to Rust. Likewise, Node.js scaling makes it a poor choice for large simulations and Python is too slow to begin with. The scaling of Rust and C++ though suggests that Rust is most competitive where it matters, beating out C++ at one million particles. Go is not that far from Rust & C/C++ and can be easier to reason about than C++ due to having a garbage collector, but the current C++ framework has no `new` or `delete` during the simulation's run, and so that advantage of Go isn't helpful in this scenario. If someone finds Go easier enough to write to justify the performance loss compared to Rust is more a point of what language one is comfortable with.

3.4 Summary

The most significant conclusion from this testing is that Rust is a competitive alternative to C++ at the scale of modern ring simulation research. Another point is that while the JVM might be tempting to consider due to its ease of multithreading and a highly optimized environment [11, 9], it is still not applicable to this scale of simulations due to the large overhead. Perhaps this could change with project Valhalla or the new Vector API. Scripting languages are even less worth considering for simulation work as their performance was much worse than the JVM's.

Chapter 4

Adaptive Time-steps and Priority Queues

4.1 Introduction

With the spring force derivations for stable soft-sphere collisions and Rust as a safe and efficient language for simulations in hand, it is time to extend this to make a full simulation. However, soft-sphere simulations still have one more obstacle to overcome, that of the time step. While a hard-sphere simulation processes a collision in a single step, a soft-sphere collision needs several steps to properly resolve the collision. As such, soft-sphere collisions need to be run at a smaller time step than hard sphere collisions. This smaller time step translates directly into a longer runtime. So to avoid this, I introduce an adaptive time step - using a priority queue. Possible collisions are put on the priority queue to be processed at some point in the future, and are then processed in sub-time steps using priority to keep events in chronological order. Thus collisions can be resolved at smaller steps than gravity, while gravity can be at the same time step as hard-sphere simulations. This is especially

important because the kD-tree based gravity algorithm is $O(n \log n)$, and so represents the most expensive part of the process. Meanwhile, particle pairs only need to be processed when they are close and might collide, and so this is a product of speed and density, not number of particles. So the calculations per particle should be roughly constant as the total number of particles varies, making this $O(n)$. In this chapter, the smaller, adaptive time step will be called sub-time step, while the time step between kD-tree gravity calculations will be called the big time step or full time step.

In addition, this simulation models the rings of Saturn, more precisely, a small window in the rings that rotates with the ring. To make the particles move as if a central body is present that they are all orbiting around, we need to add Hills force, which is the linearized solution to orbital motion in the rotating reference frame of the window we simulate. Since only a small window is being simulated, not the whole ring, particles that leave this box need to be wrapped around to stay in the box. This is accomplished using what is called a sliding brick boundary condition, that adds particles back on the other side and adjusts their velocity accordingly so that energy is not gained as the particle is moved between higher and lower parts of the central bodies' gravity well.

Advantages and Problems of a Variable Time Step Approach

The advantage of the adaptive time step system using a priority queue is that it is that the time step can adapt to the local condition - if the particles are moving very fast, the simulation will do many small steps to resolve the collision. Additionally, these many small steps are only for the pairs of particles that might collide and thus as previously mentioned the expensive $O(n \log n)$ gravity calculations don't have to be done. The downside of adaptive time steps is that the complexity is much greater, and sometimes, if the particles are moving very fast, the adaptive time steps can get so small that, when added to the current time,

they don't change the current time as they are smaller than the precision of the current time. As the time does not advance, this leads to an infinite loop. Currently, the code is set up to panic if this happens. Another thing is that if the particles move fast such that the time step is very small but not too small to lead to a true infinite loop, then the simulation can end up taking steps so tiny that it is still moving forward, but at a rate of effectively zero. This is not detectable, but so far has only happened due to another bug that was removed and thus should be inconsequential.

Since the priority queue system works on pairs, it does have problems with 3-body configurations such as a pair at rest and one particle coming in from the side, all along one axis. The incoming particle will be processed with each of the two resting ones, but the two resting ones aren't being processed since they are not approaching each other. But once the incoming particle hits the first resting one, that originally resting one will now be on a collision course with the other resting one, but since this collision was not foreseen during the walking of the kD-tree, it isn't getting processed. A workaround to this is the estimation of a global relative speed, and then rather than just looking at the speed of the particles approaching, looking instead at the max of either the speed of the particles approaching or the global relative speed estimate. This is a crude method that will lead to a lot of unnecessary sub-steps, but should deal with these types of three body problems where the velocity of one particle changes drastically in sub-steps. This is the currently implemented approach, and the implementation will be further described later.

A different idea, although not implemented here, would be to re-find neighbors that the particle might be colliding with after a significant velocity change. However, what constitutes a significant velocity change is hard to define.

4.2 Approach

4.2.1 Algorithm

The simulation is run for a full orbit, and for each step the Hills force is computed at the beginning of the step. The computed acceleration is multiplied by the full time step and immediately added to the velocity of each particle. After this completes, the kD-tree is built, and it is used to get a global relative speed estimate, which will later assist in computing how long the adaptive time steps should be. To keep this $O(n \log n)$, only the relative velocity of particles within the same leaf node are computed. This does mean that two particles approaching quickly but in different nodes won't be caught, but this should be rare, especially as the simulation gets larger and so some high speed particles will get caught, raising the speed estimate. These relative velocities are then combined using one of two methods - root mean square or maximum. For safety, the speed estimate is multiplied by 2 to minimize the impact of a high-speed particle not being caught.

Then the kD-tree is walked again. As standard for the kD-tree algorithm, for every node, the opening angle is computed and if it is below some threshold, the entire node is approximated as one particle. The gravity impulse is computed for the whole step and added to the particle's velocity. If however the opening angle is large, then the particles are too close to be approximated as one and so instead the particle currently processed is compared directly to the other particles. In this case, not only is gravity computed, but also the particles are checked for how likely they are to collide, and in what time. A sub-time step is computed, and if the sub-time step is smaller than the big time step, they are placed on the priority queue to be updated and processed at that time in the future. The particles themselves also have their velocities updated from the forces experienced, but only until they are to be processed again, not for the full time step.

The reason a priority queue was chosen is that events are added to the queue at a random order - but events need to be simulated in chronological order, and thus events are always pulled based on which has the smallest elapsed time, i.e. is closest to the current time. Once the kD-tree walk is completed and all gravity and spring forces have been computed and all collision events queued, the queue is processed. Each event is a pair of particles. Their spring and gravitational forces are computed, and a new sub-time step is computed. If the current time plus the new sub-time step is greater than the time when the next big time step occurs, then it is truncated to the next big time step. This helps synchronize all particles at the big time step as during the processing of the priority queue, each particle is at some different point in time. The acceleration from the forces are integrated for time sub-time step and the particles are put back on the priority queue, unless the truncation happens, in which case it just gets processed during the next big time step. As particles can be at different times when an event is processed, each particle is first "fast-forwarded" to the time of the event, which consists of integrating the velocity from the last time the particle was processed up to the event's time and adding that to the position. In the previous simulation, like those in chapter 3, all the acceleration of one time step are added up and then added to the velocity. This is not possible due to the sub steps taken between the big time steps. Thus all acceleration calculated is multiplied by the time until this specific source of acceleration is recomputed, and the resulting change in velocity is added to the particles' velocities. Once the queue empties, all particles are "fast-forwarded" to the time of the next big time step, and the cycle repeats.

Calculating the sub-time step

Since a collision is modeled as a spring, one full collision would be half the sprint cycle, i.e. $T/2$. By physics, $T = 2\pi/\omega_l$ and so the full collision time should be around $T = \pi/\omega_l$.

I then chose roughly how many steps a collision should take, usually around 10-20, and divide the estimated collision time by this desired step count to get the sub-time step. If the particles are currently colliding, then this is the sub-time step used. If the particles are not yet colliding, then the sub-time step is the max between this value and the time till impact, divided by 2. The global relative speed estimate is also considered. To ensure that nearby particles will be processed at a rate fast enough to account for one being slammed into and rapidly changing speed, I also compute the time it would take to travel between the particles at the global relative time step. The final sub-time step is the minimum of the two.

4.2.2 Initial Priority Queue Validation

Before beginning with large simulations, I wanted to verify the priority queue method with a simple particle-particle test, in the style of the work done in chapter 2. Particles are set up symmetrically, approaching each other with some set impact velocity. Once the particles separate, the simulation is terminated and the coefficient of restitution and maximum penetration depth are collected. I tested a variety of parameters, detailed below:

- desired coefficient of restitution: $0.2, 0.5, 0.7, 0.9$
- time step: $1., 0.5, 0.1, 0.05, 0.025,$
- particle radii: `1e-7, 3e-8, 1e-8`
- ratios of particle radii: 1:1, 1:3, and 1:10
- impact speed: `3e-8, 1e-7, 3e-7, 1e-6, 3e-6`
- desired sub-time steps during one collision: $1, 3, 5, 7, 10, 12, 15, 20$

The density of particles was modeled after those in Saturn's B rings. As most of these time steps are so large that the particles would just pass through each other, the effectiveness of the priority queue system was required for any sensible collision resolution. The parameters are chosen in a way to reflect the work done in chapter 2. From the three derivations presented in that chapter, I will use my own derivation, labeled Rotter. As the particles are heading straight for each other, gravity was turned off to have greater control over the impact velocity and focus just on the spring restoring force model for collisions.

4.2.3 Full Simulation Performance Benchmarking

To check whether the adaptive time step with a priority queue is actually faster, a benchmark of the full simulation with the sliding brick boundary conditions and Hills Force both with and without the priority queue, across various simulation sizes and particle densities corresponding to Saturn's A and B rings is needed. However, due to time constraints, I wasn't able to add the sliding brick boundary conditions as they are quite unforgiving in soft-sphere simulations. As particles are wrapped around, they can be placed on top of another one, causing the two to "blow up", which ruins the accuracy of the whole simulation. In practice, "mirrors" are used where particles on the borders interact with mirrors of themselves, and so a particle would interact with the mirror of another before the other particle is wrapped over. Due to time constraints I won't be able to setup this however, and so instead I will simulate a tall rectangle, so the shearing due to the Hills Force is lessened in effect. Particles closer to the imagined center will orbit faster, and those further out will orbit slower, and so a rectangle of particles left on its own will shear out in orbit.

Each simulation case will be benchmarked 5 times on an Apple M1 Pro 3.2 GHz (the version with 10 CPU cores) and the average will be reported. For the simulations with the adaptive time step, I will use a time step similar to those used for hard-sphere collisions,

which is $\frac{2\pi}{1000}$ (2π is one orbit, so this is 1000 steps per orbit). For the soft-sphere collisions without the adaptive time step, a smaller time step is needed to keep the simulation numerically stable, requiring a time step of $\frac{2\pi}{10000}$. Each simulation will be run for 1/10 of an orbit, which is 100 steps for the adaptive time step simulations and 1000 for those without.

4.3 Results

4.3.1 Initial Priority Queue Validation

The results of the initial priority queue validation can be seen in figure 4.1. For the data points targeting the coefficient of restitution of 0.5, those can be seen in the center of the graph. As the desired step count in a collision increases (the right side of a particle bunch), the measured coefficient of restitution approaches the target 0.5. So the greatest two desired step counts tested, 15 and 20, seem to produce quite good results on a variety of impact velocities (dot size in the scatter plot) and even produces good results when the particles are mismatched in size. The penetration depth is also quite nicely constrained, not varying too much from 1%. For the full simulation, I will thus use 15 as the desired step count.

Looking back at chapter 2, specifically figure 2.3 and 2.1, the soft-sphere model on its own struggles with larger impact speeds or particles of different sizes. However, since with this adaptive time step method I can pick how many steps I want in a collision, the results in figure 4.1 for ten or more steps vary much less as the particles' radii differs. In figure 4.2, it can also be seen that the impact speed doesn't have much of effect on the final coefficient of restitution and maximum penetration depth is also well constrained for larger impact speeds. Each group in this figure is sorted by desired collision steps, with the larger step counts on the right, and so the more scattered output variables are found on the left side, where the simulation didn't get enough steps to resolve the collision. Thus this adaptive

time step method is not just promising for speeding soft-sphere simulations up, it also helps increase their numerical stability. When comparing the figures of chapter 2 and the figures in this chapter, it is also worth noting the difference in scaling: the scale of the penetration depth seen is 10x smaller with the adaptive time step approach. For the coefficient of restitution, the scatter range also decreased - from up to 0.8 in chapter 2 to only up to 0.6 with the adaptive time step (Where again, we are aiming for 0.5).

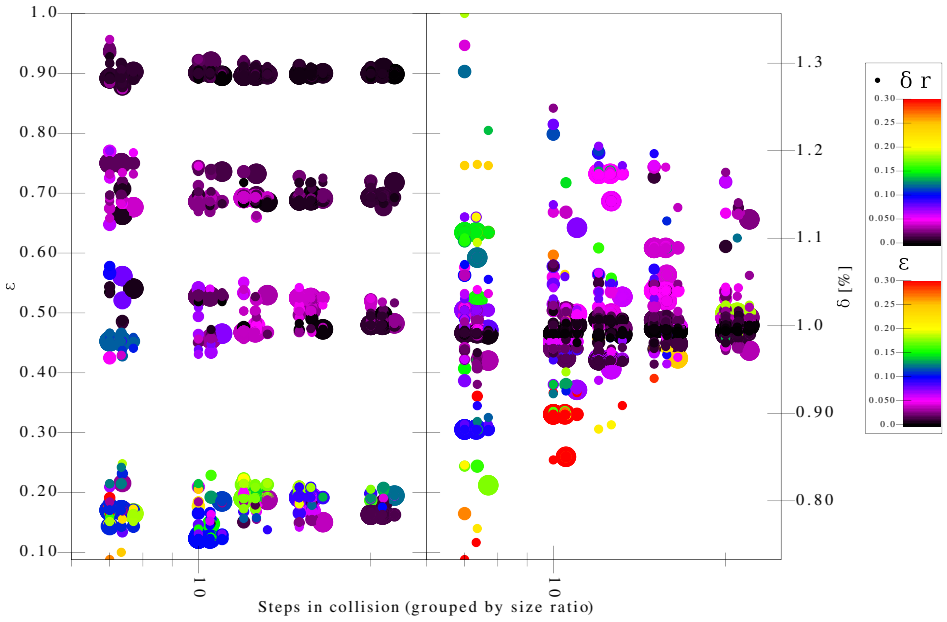

Figure 4.1: The coefficient of restitution is plotted on the left, and the maximum penetration depth on the right. For each side, the colors indicate the other variable. The desired step count in a collision, used to compute the adaptive time step is the x-axis. Within one bunch of particles, left-to-right represents an increase in the ratio of radii of the pair of particles colliding, the leftest being 1:1, 3:1, and finally 10:1 on the right. The size of a data point represents impact velocity

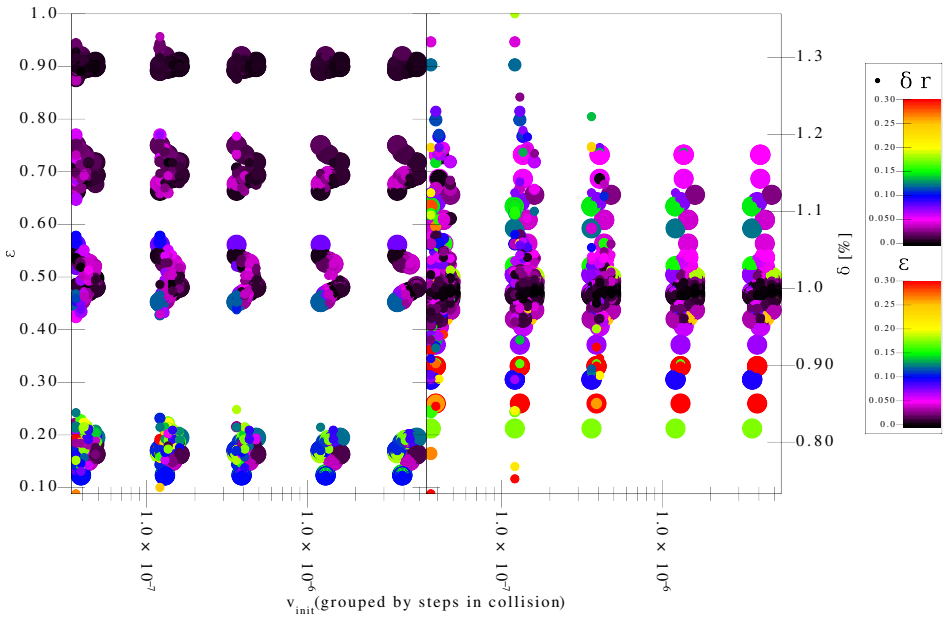

Figure 4.2: The coefficient of restitution is plotted on the left, and the maximum penetration depth on the right. For each side, the colors indicate the other variable. The impact velocity is on the x-axis. Within one bunch of particles, left-to-right represents an increase in the desired collision steps.

4.3.2 Full Simulation Performance Benchmarking

The results of the benchmarks that were successful are shown in table 4.1. Given that the kD-tree algorithm is $O(n \log n)$, as the particle count scales by 10, the time should scale by 12-13.

The benchmark values speak well of the adaptive time step with a priority queue approach, averaging at about 8 times faster than the approach without a priority queue.

Additionally, there is no downwards trend in improvement as the simulation scales, which is important as astrophysics simulations often use much more particles than the particle counts I benchmarked. Instead, it seems like there is an upward trend. See figure 4.3. It is also worth noting that since density increases interactions between particles as they attract more strongly, density has an effect on the runtime. This can be best seen in the difference between A-ring density and B-ring density. In the case of the priority queue, more sub-steps are computed when interactions are more intense, but clumping also leads to a slowdown in the kD-tree as more gravitational interactions become particle-particle rather than particle-node, which means less time-saving approximations are made.

	Priority Queue		No Priority Queue	
	A Ring	B Ring	A Ring	B Ring
1k	0.52	0.48	4.3	3.5
10k	24.8	18.6	149.5	100.3
100k	405.9	355.0	5248.1	3091.0
1M	6617.2	5966.3		75796.3

Table 4.1: All measurements are in seconds. The A & B refers to the particles scaled density setting, as particles in Saturn's A ring are almost twice as dense as those in the B ring in the scaled units used. They are the same density in terms of g/cm^3, but the scaled coordinates also capture the effect of tidal forces, so A-ring particles clump more than B-ring particles. The 1M run without the priority queue for A-ring particles couldn't be tested due to time constraints as the runs are so long.

4.4 Summary

While the results so far are encouraging and make it possible to run bigger and longer soft-sphere simulations, it would be optimal to run the benchmarks with the sliding brick boundary conditions. Sliding brick boundary conditions seem to be difficult to keep numerically stable in soft-sphere as any overlap between particles greater than 6% usually spells

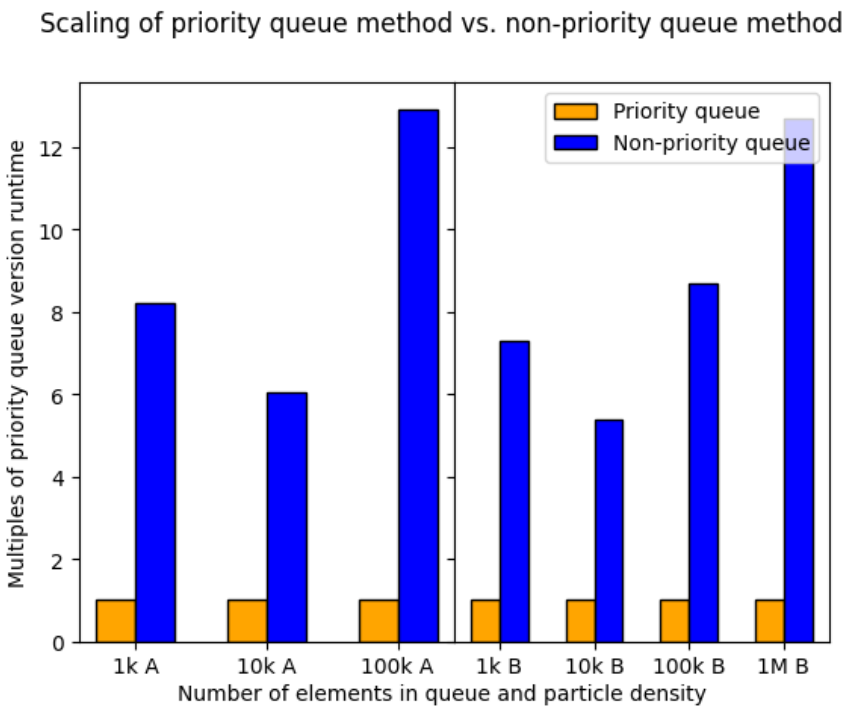

Figure 4.3: Execution times as a multiple of the priority queue version of that particle count and density.

the end of a simulation and so great care is needed to ensure that never happens when particles are wrapped over. Thus further research is needed in using boundary conditions in soft-sphere simulations.

Since the simulation without the adaptive time step uses a timestep 10x smaller, the best possible speedup seen by the adaptive time step method should be 10x. However, there are a few runs with 12x speedup, which is another point of future research. One possibility is that the difference in numerical stability over time leads to significantly different kD-trees being built.

Besides this, this simulation code could use some ironing-out of edge cases as during the benchmarks, overlaps of up to 4% were seen and optimally, overlaps should stay in the $1 - 2\%$ regime. However, compared to the penetration depths of up to 30% that was seen in the tests of chapter 2, these results are much better and so represent a significant step forward in numerical stability and physical accuracy of soft-sphere simulations.

Chapter 5

Conclusion

With the derivations of the spring constant and damping in chapter 2, soft-spheres can now be included in astrophysics simulations, however at the cost of a significantly smaller time step, i.e. a factor of 10. As per chapter 3, Rust is competitive to C++ in terms of speed and memory usage, and with its memory safety, it would make a good modern language for simulation. In chapter 4, I experimented with the use of an adaptive time step using a priority queue. While more work is needed to properly implement boundary conditions, soft-sphere simulations with the adaptive time step method are much faster than soft-sphere simulations without, and are also much better at matching the desired coefficient of restitution and penetration depth, thus promising soft-sphere simulations with much better numerical stability than those without the adaptive time step method.

Bibliography

[1] Anon. The computer language 22.05 benchmarks game, 2021.

[2] Anon. Sustainable scala, 2021.

[3] S. Araki and S. Tremaine. The dynamics of dense particle disks. *Icarus*, 65(1):83–109, January 1986.

[4] Frank G Bridges, A Hatzes, and DNC Lin. Structure, stability and evolution of saturn's rings. *Nature*, 309(5966):333–335, 1984.

[5] AP Hatzes, F Bridges, DNC Lin, and S Sachtjen. Coagulation of particles in saturn's rings: Measurements of the cohesive force of water frost. *Icarus*, 89(1):113–121, 1991.

[6] Piet Hut, Jun Makino, and Steve McMillan. Building a better leapfrog. *The Astrophysical Journal*, 443:L93–L96, 1995.

[7] Mark Lewis Jonathan Rotter. Soft body collisions for ring simulations with rust. 2021.

[8] Mark Lewis Jonathan Rotter. N-body performance with a kd-tree: Comparing rust to other languages. 2022.

[9] Jason Leezer, Mark Lewis, and Berna L Massingill. A java based framework for numerical simulations of collisional systems. In *PDPTA*, pages 297–303, 2008.

[10] Mark Lewis. Hard sphere coll sim. `https://github.com/MarkCLewis/HardSphereCollSims`.

[11] Mark Lewis and Berna L Massingill. Multithreaded collision detection in java. In *PDPTA*, pages 583–592, 2006.

[12] Mark C. Lewis and Jonathan Rotter. Multi-language kd-tree n-body benchmarks, 2022.

[13] Mark C Lewis and Glen R Stewart. A new methodology for granular flow simulations of planetary rings–coordinates and boundary conditions. In *Proceedings of the IASTED International Conference, Modeling and Simulation*, pages 292–297. ACTA Press, 2002.

[14] Mark C Lewis and Glen R Stewart. A new methodology for granular flow simulations of planetary rings-collision handling. In *Modelling and Simulation*, pages 292–297, 2003.

[15] Mark C Lewis and GR Stewart. Collisional dynamics of perturbed planetary rings. i. *Astronomical Journal*, 120(6):3295, 2000.

[16] NIST. Safer languages, 2023.

[17] Stephen O'Grady. The redmonk programming language rankings: January 2022, 2022.

[18] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[19] The PyPy Project. Differences between pypy and cpython, 2022.

[20] The PyPy Project. Downloading and installing pypy, 2022.

[21] The PyPy Project. How fast is pypy3.9?, 2022.

[22] The PyPy Project. Modified n-body speed comparison, 2022.

[23] Derek C Richardson, Kevin J Walsh, Naomi Murdoch, and Patrick Michel. Numerical simulations of granular dynamics: I. hard-sphere discrete element method and tests. *Icarus*, 212(1):427–437, 2011.

[24] H. Salo, K. Ohtsuki, and M. C. Lewis. *Computer Simulations of Planetary Rings*, pages 434–493. 2018.

[25] Heikki Salo. Simulating the Formation of Fine-Scale Structure in Saturn's Rings. *Progress of Theoretical Physics Supplement*, 195:48–67, 07 2012.

[26] Stephen R Schwartz, Derek C Richardson, and Patrick Michel. An implementation of the soft-sphere discrete element method in a high-performance parallel gravity tree-code. *Granular Matter*, 14(3):363–380, 2012. In a leapfrog, the velocity is out of sync and so since it is used for some damping forces, it is naively predicted. They also did twisting and rolling forces. k is chosen in regard to the max velocity. Warnings are emitted if penetration is too big. Tangential computation is important for lasting contacts. Equations for all contact forces.

[27] J. Wisdom and S. Tremaine. Local Simulations of Planetary Rings. *Astronomical Journal*, 95:925, March 1988.

Appendix A

Code

All code can be found in various Github repositories.

- Chapter 2: `https://github.com/MarkCLewis/RustCollSim/tree/jonathan`
- Chapter 3: `https://github.com/MarkCLewis/MultiLanguageKDTree`
- Chapter 4: `https://github.com/MarkCLewis/RustCollSim/tree/pq-start`