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

1st Jonathan Rotter
Department of Computer Science
Trinity University
San Antonio, TX, USA
jrotter@trinity.edu

2nd Mark C. Lewis
Department of Computer Science
Trinity University
San Antonio, TX, USA
mlewis@trinity.edu

***Abstract*—This paper presents the results of a more representative N-body benchmark utilizing a kD-tree implemented in multiple languages. We find that while Rust is slightly slower than C or C++ for smaller simulations, it is the fastest language for simulations at the scale we use in actual research. On the other hand, Go is constantly 1.5-2x slower than Rust. The JVM is competitive with Go for intermediate-size simulations but struggles when we reach one million particles. As expected, scripting languages are the slowest, though it is rather remarkable how much slower Python is than Node.js. Also surprising is that our attempts to speed up the Python implementation using NumPy made it significantly slower.**

***Index Terms*—simulation, performance, n-body, kD-tree, Rust**

I. Introduction

The Benchmark Game is part of a 20+ year effort to create a set of simple performance benchmarks across various languages [1]. Over that time, these benchmarks have become a reasonably well-known and accepted comparison of language performance, given the provisos, they note in their discussion. Recently, these benchmarks have received renewed interest as the basis for various academic papers comparing energy efficiency across languages [2], [3].

Fairly early in the history of these benchmarks, one of the authors of this paper proposed the N-body benchmark and submitted the original implementations for Java and C++. This code does a standard O(n^2) calculation of gravity for the planets in our solar system. That N-body approach is a good benchmark for basic number crunching. However, most modern N-body simulations involve a much larger number of particles and use more complex methods, generally involving data structures, to do the force calculations in O($n \log n$) or O(n) time. This paper aims to look at performance across a number of languages using a more realistic computational approach.

We are particularly interested in Rust's performance as an alternative to C/C++ for these types of simulations [4]. Rust is a more modern language with greater expressivity and a type system that provides strong code quality guarantees, so there is a general interest in being able to use it for astrophysical and HPC work [5], [6].

II. Approach

The most commonly used method for speeding up N-body calculations is using spatial trees as introduced initially by [7]. The idea of the tree codes is that the gravitational interaction between a particle and others far away from it are well approximated by treating the collection of distant particles as a single unit. The first-order approach is to calculate the force that a single point mass would exert at the center of mass of the grouping. More accuracy can be derived by considering the higher-order terms in a multi-pole expansion [8].

The original work by Barnes and Hut used an octree as their spatial data structure, where each node was divided into eight regions of equal size by splitting in the middle along each of the three spatial axes. More recent work has often used a kD-tree instead [9], [10]. In a kD-tree, each internal node has two children, and the split between them is at a specified value perpendicular to a given axis. So the kD-tree is effectively a binary tree for high-dimensional spaces. There are many ways to choose the split parameters for each node in a kD-tree. For this work, we use the approach of splitting at the median along the dimension with the largest separation between bodies. One significant advantage of the kD-tree over the octree is that the tree is always balanced by splitting at the median.

A rendering of a kD-tree from one of our simulations is shown in figure 1. The system we simulate for these tests is a disk of small particles in orbit around a central mass. The particles are placed on circular orbits and are uniformly distributed in distance from the central mass and radial location. In many ways, it is like a low-mass particle disk in our Solar System going from 0.1-5.0 AU with a variable number of particles. The green dots in fig 1 are the locations of bodies in that particular simulation. Our kD-trees were all constructed with a maximum of seven particles in each leaf. The complete code used for this work, with all the details on parameters and approach, can be found at [11].

A. Language Selection

The Benchmark Game has had years to accrue a broad range of contributions across various languages from those interested in improving the performance of particular languages and platforms. Given time and knowledge constraints, we had to be somewhat more selective in our choices. We are very open

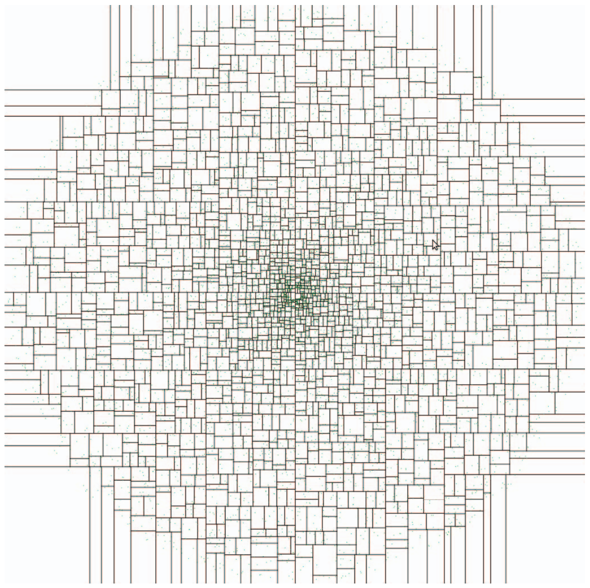
Fig. 1. This image shows the kD-tree for a simulation with 10,000 particles after 90 time steps.

to contributions from others that will add to the language selection or improve performance in our languages. Our goal was to include most of the languages/platforms present in the most recent RedMonk programming language ranking [12]. Following the lead of the Benchmark Game, we include only one language per platform. So we have TypeScript instead of JavaScript and Java but not Scala or Kotlin. The argument is that optimized versions should run at the same speed in all languages on a single platform[1]. More idiomatic code in various languages might indeed produce different performance. The Scala Center [13] saw this behavior in their work using the methodology from [2].

While we did not test multiple languages on the same platform, we did test some variations in some languages. For example, in Rust, we tried using their explicit SIMD library in addition to a version without explicit SIMD. In Java, we wrote a "standard" object-oriented version using a mutable class for particles and a version that controls the memory layout more by putting all the coordinates in large arrays.

Another implication of using many languages is that our code in all languages is single-threaded. This is primarily because not all of the languages in our sample include good support for multithreading. Python and JavaScript, particularly, have weaker support than the other languages. The other reason to leave out multithreading at this point is because of the complexity that it adds. While this benchmark has to be more complex than the O(n^2) code in the Benchmark Game, we did strive to not add additional complexity beyond using the kD-tree to simplify porting it to various languages. While the acceleration calculations are embarrassingly parallel, the tree building is not. Indeed, the approach to tree building implemented in this code is greatly simplified by the fact that it is inherently single-threaded.

[1]This might not be true for languages like Clojure, because they use dynamic typing despite being on a platform with statically typed, compiled languages.

B. You can write Fortran in any language

Contrary to the advice of [14], there are several ways in which the code we have written resembles Fortran in all the languages used. Most notably, we use arrays and indexing instead of dynamic memory whenever possible to minimize memory allocation. Reducing memory allocations provides speed benefits in all implementations. It also makes translation between languages easier.

Another reason for this style choice is that it is what had been used in a more general simulation framework written in C++ that we were initially comparing the performance of our Rust implementations to [15], [16]. This code base is much more advanced, but it gave us a good sanity check for performance when building the Rust version, which was the version we created first for this work. We then used the Rust version as a template for converting to all the other languages to keep implementations reasonably consistent.

III. Results

We ran performance tests on a workstation with two Intel®Xeon®E5-2680 v3 CPUs and 64 GB of RAM. To remove any variability between how language libraries report timing, we used the Linux `time` command and collected the user time. We did this for simulations of various sizes ranging from 1000 particles up to one million particles. We ran all simulations for 100 iterations and repeated them five times. Table I shows the timing results means and standard deviations.

TABLE I
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 and C++, we initially used GCC. However, as Rust uses an LLVM-based compiler, we decided to try Clang to see how it might differ. Clang was somewhat faster ($< 10\%$) than GCC, so the table shows the Clang results. We have omitted a time for Python at one million particles because the execution times would be on the order of a week. The Java environment was GraalVM 22.2.0 for Java 17.0.4, we ran TypeScript using Node v10.19.0, and Python used CPython 3.10.7.

Figure 2 shows a chart of the execution times as multiples of the standard Rust version for each size simulation. Python has been left out of the graph because the times for Python were so large that they made it impossible to see the details of the other languages. There are a few things that stand out in this figure. Some of them are things we might expect like the fact that Rust, C, and C++ are faster than the other languages. Similarly, Go and the JVM are faster than Node.js. In the smallest runs, with only 1000 particles, the startup time for the JVM makes it almost as slow as Node.js. At the mid sizes, the JVM roughly ties Go, but Go is faster at the smallest and largest sizes. What might be surprising to some who have not seen these types of results before is that Python is so much slower than Node.js.

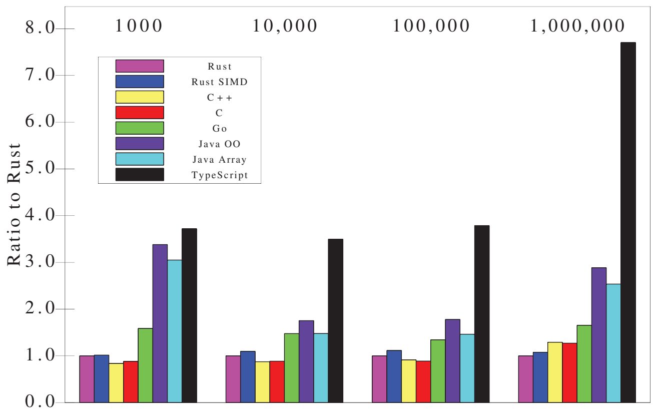

Fig. 2. Execution times as a multiple of the Rust execution time for that simulation size.

A. Memory Matters

Memory usage is critical on modern computer hardware. Fetching memory from RAM takes ~100 clock cycles while pulling it from the cache is roughly 10x faster. For this reason, using memory in a manner that is cache friendly can significantly benefit performance. The Xeon®E5-2680 v3 used for these benchmarks has the following cache details.

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

The shared vs. separate cache distinction is unimportant because this code is single-threaded.

The importance of memory size and layout appears in many ways in these results. The memory layout is significant for the two versions of Java, while memory size plays a significant role for Rust. To help understand the role of memory in these benchmarks, we measured memory usage for each of the various simulations. Memory usage was measured with the GNU `time` command with the `-v` option. Table II presents the "Maximum resident set size" from that output. Note that all values are greater than 1MB, so none of these can fit completely in L1 or L2 cache on our test platform. However, all the Rust, C++, and C simulations with fewer than one million particles can fit in the L3 cache.

TABLE II
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
Golang	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	–

This table shows a clear break in memory usage, with Rust, C++, C, and Go being far more memory efficient than the other languages. The JVM is notorious for having a large memory footprint, which clearly shows here, though Node.js is not much better.

These languages have larger memory footprints because of their virtualized environments. They also have memory models that make it hard to control where values are in memory, so caching behaviors can be poor. For example, compare the `Particle` type in both C++ and Java in the GitHub repository. They look very similar, with positions and velocities in arrays of three elements and single values for radius and mass. However, in C++, this compiles to a single chunk of memory big enough to hold the combined eight doubles. In comparison, this is one object with two references and two doubles in Java. Those references point to array objects with three doubles and a length, plus some additional overhead. Similarly, the whole system is stored as a collection of `Particle`. In C++, that `vector<Particle>` becomes a single large chunk of contiguous memory. In Java, the `ArrayList<Particle>` stores an array of references that point to the object just described. So the object-oriented Java implementation might look like that for Rust, C++, and C in code, but it has a very different layout in memory with a lot more overhead in terms of actual memory footprint and following references during runtime.

To minimize this, we wrote a version that creates large arrays of `double` to store positions, velocities, radii, and masses. In this implementation, there are only four objects, and because of how Java treats arrays of primitives, each has a single, large chunk of memory. This approach is consistently 10-20% faster than the object-oriented approach. One thing that might stand out in the memory usage chart is that this approach had the largest memory footprint. We speculate that this is because the reported value is the largest resident memory set. At the point where the garbage collector copied these large arrays from short-term memory in the GC pool to a longer-term segment of memory, they were both active and resident for a while.

On the Rust side, it might seem odd that using SIMD can slow things down, but that is what our results consistently show. We speculate that this is because our vectors only need 24-bytes for three double-precision floating point values. The

SIMD values include 32-bytes for four values. This explains why the SIMD version of Rust has a larger memory footprint for all simulation sizes. The SIMD instructions also work on those extra values, but if they are done with SIMD instructions, that should not provide additional overhead. Indeed, in independent testing of SIMD instructions on an implementation of the simpler, O(n^2) N-body code, we find that SIMD does indeed speed things up. What we see in this work is that the additional memory overhead slows things down more than the SIMD instructions speed them up.

B. Scaling

As stated earlier, using a tree for gravity calculations should give us code that scales an O($n \log n$). Based on that scaling, a naive prediction of the runtime ratios when the particle count is increased by 10x would be in the 12-13 range for the sizes we are testing. Figure 3 shows the actual values for each language. This figure shows that, in practice, the runtime ratios are more in the 18-20 range. This value is consistent for 1000 to 10,000 particles and 10,000 to 100,000 particles, except for the JVM implementations, where the 1000-particle runs were skewed by JVM startup.

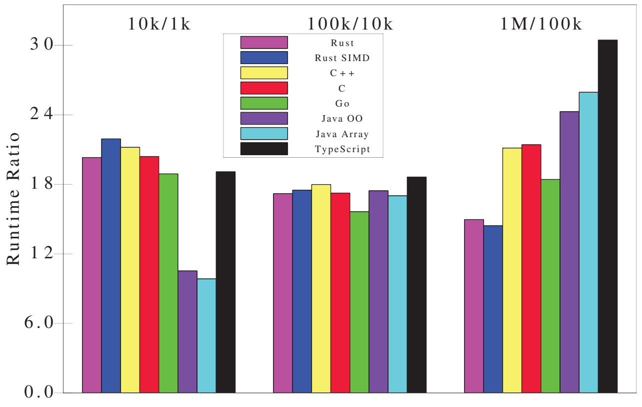

Fig. 3. Execution time scaling with particle count.

Going from 100,000 to one million particles, we see many more interesting behaviors that vary across the languages. The jump to one million particles makes all the simulations use more memory than the L3 cache on the benchmark platform. For that reason, we might expect that the increase in runtime will be larger going from 100,000 to 1,000,000 particles than for other jumps. Indeed, this is true for all languages except Rust. We do not yet understand why Rust does so well with a larger number of particles, but it inevitably deals with details of how the code is accessing memory. This good scaling makes Rust the fastest language for the largest simulations. It is also interesting to note that the JVM and Node.js scale very poorly in this last jump and are particularly slow for the largest simulations. We cannot say anything about what happens with Python at this step because the runtime for Python for one million particles is so long that it is impractical to benchmark.

C. Python is Horribly Slow

That leads to the next thing that stands out in these benchmarks. Python is exceptionally slow. To clarify how Python compares to the other languages, figure 4 shows the same data as figure 2, including Python and excluding the largest simulation, for which we did not benchmark Python. One thing that this figure shows that the table didn't make clear is that Python is getting better relative to the faster languages as the simulations get larger. However, at every size tested, it is well over 100x slower than Rust.

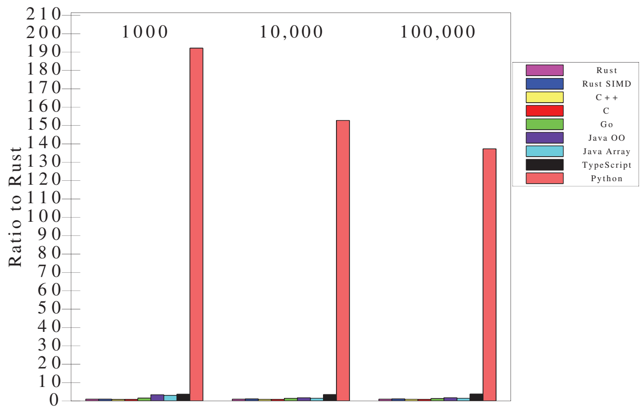

Fig. 4. Execution times as a multiple of the Rust execution time for that simulation size including Python.

Every other language finished the one million particle simulation faster than Python finished the 100,000 particle simulation, even Node.js. This result is consistent with the timing results in the Benchmark Game for the n-body benchmark. Their only solution for Python took 9 minutes, which is over 100x slower than the fastest solutions.

The implementation tested was written in pure Python. We made several attempts to use NumPy, a library for efficient arrays and operations on them, to improve execution time. One set of attempts used NumPy arrays for the positions and velocities in particles. Another set of attempts used single large NumPy arrays similar to the faster Java implementation. Everything we have tried has resulted in ~2x worse performance than the pure Python implementation. The fact that NumPy does not improve performance is consistent with the Python submissions in the Benchmark Game being written in pure Python, which indicates that others have not found a way to improve the performance with NumPy or other numerical libraries. While NumPy excels at operating on arrays, the kD-tree approach results in significant time being spent on traversing the tree. In fact, computing the acceleration by traversing the tree is an $O(n \log n)$ operation and represents the majority of the computational work while integration, which can easily be optimized with NumPy, is only $O(n)$. As such, the kD-tree algorithm does not play into NumPy's strengths.

One possible speedup for Python would be using Cython, a superset of Python that easily integrates C with Python.

For example, we could write the kD-tree in a Cython or C extension. However, if most of the logic happens in C, it is not fitting to still call it Python's performance.

Each new version of Python adds speed improvements. For example, Python 3.11 is about to be released at the time of writing, and they claim a 1.25x speed improvement [17]. Putting that into perspective, it means Python is 100x slower than Rust instead of 125x slower. These incremental performance boosts are helpful but insufficient to make Python a competitive language for numerical work without most of the actual code being written in a different language.

D. PyPy to the Rescue?

An alternative to the standard implementation, CPython, is PyPy. PyPy is a JIT compiler that, on average, can reach 5x the speed of CPython [18]. Preliminary testing suggests that for our kD-tree code, PyPy runs 10-15x faster than CPython. Since our pure Python codebase does not rely heavily on C functions, the JIT compiler can optimize most of the computations. Our results for PyPy were even more impressive than other benchmarks against CPython3.7 using a modified N-body problem, where a 6.5x speedup was seen compared to CPython3.7 [19].

While PyPy provides speed improvements, there are other challenges using PyPy. Our original Python implementations used features of Python 3.10. The code had to be modified to use PyPy because the current implementation of PyPy is for Python 3.7 [20]. In addition, the speed benefits of PyPy come not only from using a JIT but also from semantic differences from standard Python as implemented in CPython [21]. Because of the JIT, PyPy uses significantly more memory than CPython. PyPy's memory usage was similar to the JVM in our testing. So while PyPy is much faster than CPython, using it comes with some costs, and it is worth pointing out that PyPy is still 3x slower than Node.js and 10-20x slower than Rust. So being faster than CPython does not mean it is competitive with other languages.

E. Performance Where it Matters

While there are many interesting details in these results related to how performance scales with particle count, it is worth noting that in practice, the simulations this benchmark is based on tend to involve at least a million particles. For that reason, the result that matters the most is for the largest runs done here. All the others are smaller than our typical research simulation workloads. Given this fact, there are two key takeaways from this work. First, Rust seems to be the fastest language when it counts, with or without explicit SIMD. Second, using the JVM languages might seem okay if the systems are smaller, but the performance gap grows significantly for the systems we want to study. Using Node.js or Python for this type of work should probably never be viewed as an option.

We have not said much about the Go language yet. Perhaps because its behavior is so consistent in the benchmarks, it is consistently 1.5-2x slower than Rust. Go also scaled vary consistently from one simulation size to the next. Although the garbage collector in Go can make it easier to code and reason about than C or C++ in many situations, our C++ code has no explicit calls to `new` or `delete` which makes this somewhat moot here. The Rust language does not include GC, but the way it works removes explicit memory handling from the majority of programs. Whether one finds Go to be a sufficiently more productive language than Rust to validate the decrease in speed is a matter of experience and preference. We found that the conversion to Go from the original Rust version had challenges because while Go has a garbage collector, there is explicit syntax for passing things by reference that must be considered. This explicit reference handling led to some bugs that had to be tracked down. Of course, more experienced Go developers would likely think of this and handle it more quickly. It is interesting to note that the types of bugs we experience with the Go implementation would be syntax errors caught by the type system in Rust.

IV. Conclusions and Future Work

For us, the most significant conclusion of this work is that Rust is a viable alternative for numerical work at the scale desired for modern research. It also makes it clear that neither the JVM, nor scripting languages, are really appropriate for astrophysical numerical simulations. In general, the JVM is a highly optimized environment, and the ease of multithreading makes it temping to consider [22], [23], but there is simply too much overhead associated with the platform for numerical work. Perhaps project Valhalla or the new Vector API could change this somewhat, but this work casts doubt on even that.

Looking forward, this work could be extended in a number of ways. The most obvious is that we would drop the scripting languages and add parallelism through multithreading. This is straightforward for the force calculations for all the languages used here, but parallelizing the tree building is significantly more challenging and finding an approach that works well across all the languages is an interesting problem.

Another area that we did some exploration of is looking at these benchmarks on other architectures as the Xeon used in this work is a bit older. We did some preliminary work with the ARM M1 used in newer Macs and found one potentially interesting result that the advantage for Rust at a million particles seemed greater on the M1 than on the Xeon. The comparison between architectures becomes even more interesting once parallelism is included and pitting ARM against both Xeon and EPYC architectures for various languages could have interesting implications for decisions on hardware purchases.

References

[1] Anon. (2021) The computer language 22.05 benchmarks game. [Online]. Available: https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[2] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva, "Ranking programming languages by energy efficiency," *Science of Computer Programming*, vol. 205, p. 102609, 2021.

[3] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages," in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, 2017, pp. 1–8.

[4] A. Hansen and M. C. Lewis, "The case for n-body simulations in rust," in *The 2016 International Conference on Scientific Computing*. CSREA Press, 2016, pp. 3–7.

[5] S. Blanco-Cuaresma and E. Bolmont, "What can the programming language rust do for astrophysics?" *Proceedings of the International Astronomical Union*, vol. 12, no. S325, pp. 341–344, 2016.

[6] M. Costanzo, E. Rucci, M. Naiouf, and A. De Giusti, "Performance vs programming effort between rust and c on multicore architectures: case study in n-body," in *2021 XLVII Latin American Computing Conference (CLEI)*. IEEE, 2021, pp. 1–10.

[7] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[8] J. G. Stadel, *Cosmological N-body simulations and their analysis*. University of Washington, 2001.

[9] D. C. Richardson, T. Quinn, J. Stadel, and G. Lake, "Direct large-scale n-body simulations of planetesimal dynamics," *Icarus*, vol. 143, no. 1, pp. 45–59, 2000.

[10] J. Stadel, J. Wadsley, and D. C. Richardson, "High performance computational astrophysics with pkdgrav/gasoline," in *High Performance Computing Systems and Applications*. Springer, 2002, pp. 501–523.

[11] M. C. Lewis and J. Rotter. (2022) Multi-language kd-tree n-body benchmarks. [Online]. Available: https://github.com/MarkCLewis/MultiLanguageKDTree

[12] S. O'Grady. (2022) The redmonk programming language rankings: January 2022. [Online]. Available: https://redmonk.com/sogrady/2022/03/28/language-rankings-1-22/

[13] Anon. (2021) Sustainable scala. [Online]. Available: https://www.scala-lang.org/blog/2021/12/14/sustainable-scala.html

[14] D. Seeley, "How not to write fortran in any language: There are characteristics of good coding that transcend all programming languages." *Queue*, vol. 2, no. 9, pp. 58–65, 2004.

[15] M. C. Lewis and G. 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*. ACTA Press, 2002, pp. 292–297.

[16] ——, "A new methodology for granular flow simulations of planetary rings-collision handling." in *Modelling and Simulation*, 2003, pp. 292–297.

[17] Anon. (2022) What's new in python 3.11. [Online]. Available: https://docs.python.org/3.11/whatsnew/3.11.html

[18] T. P. Project. (2022) How fast is pypy3.9? [Online]. Available: https://speed.pypy.org/

[19] ——. (2022) Modified n-body speed comparison. [Online]. Available: https://speed.pypy.org/comparison/?exe=2%2B2360%2C9%2B2414%2C9%2BL%2Bdefault&ben=8&env=3&hor=true&bas=9%2B2414&chart=normal+bars

[20] ——. (2022) Downloading and installing pypy. [Online]. Available: https://doc.pypy.org/en/latest/install.html

[21] ——. (2022) Differences between pypy and cpython. [Online]. Available: https://doc.pypy.org/en/latest/cpython_differences.html

[22] M. Lewis and B. L. Massingill, "Multithreaded collision detection in java." in *PDPTA*, 2006, pp. 583–592.

[23] J. Leezer, M. Lewis, and B. L. Massingill, "A java based framework for numerical simulations of collisional systems." in *PDPTA*, 2008, pp. 297–303.