FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Lookup-Table based Force Calculations for Molecular Dynamics Simulations with AutoPas

Jan Hampe

FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Lookup-Table based Force Calculations for Molecular Dynamics Simulations with AutoPas

Lookup-Tabellen basierte Kraftberechnungen für Molekulardynamik Simulationen mit AutoPas

Author:	Jan Hampe
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Markus Mühlhäußer, M.Sc., Fabio Gratl, M.Sc
Date:	15. April 2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15. April 2024 Jan Hampe

Acknowledgements

I thank my two advisors Markus Mühlhäußer and Fabio Gratl for their supervision and support in matters concerning this work. Furthermore I thank Prof. Dr. Hans-Joachim Bungartz for the opportunity to work at their chair on such an interesting project. Lastly I thank my girlfriend for her endless support.

Abstract

In this work the prospect of utilizing lookup tables in the molecular dynamics simulator md-flexible, which is part of the AutoPas project, is investigated. First the neccessary theoretical background on molecular dynamics simulations, lookup tables and the Lennard-Jones and Axilrod-Teller potentials is established. Then different implementations to efficiently compute these are discussed and investigated for their space-efficiency and performance characteristics.

Zusammenfassung

In dieser Arbeit wird die Möglichkeit Lookup Tabellen in dem Molekulardynamik Simulator md-flexible, welcher Teil des AutoPas Projekts ist, untersucht. Zunächst wird der nötige theoretische Hintergrund bezüglich Molekulardynamik Simulationen, Lookup Tabellen sowie den Lennard-Jones und Axilrod-Teller Potentialen erörtert. Dann werden verschiedene Implementationen, um diese effizient zu berechnen, diskutiert und ihre Speicherplatz- sowie Geschwindigkeitscharakteristika untersucht.

Contents

Part I.

Introduction and Theoretical Background

1. Molecular Dynamics Simulations

> "Computational simulations and thus scientific computing is the third pillar alongside theory and experiment in todays science." [RWK$^+$09]

By no means are simulations able or supposed to permanently replace either theoretical analysis nor real-world experiments, but they contribute greatly to the toolkit that is available to those seeking to advance their scientific field of choice [BZBP13]. They have therefore become an integral part in the scientific process of discovering, interpreting, contextualizing and validating new knowledge.
The definition of "simulation" can by no means be considered agreed-upon, but one can very inclusively define it as any process or techology concerned with the prediction or recreation of a certain scenario. [BZBP13]. Simulations of real-world processes were some of the first tasks given to electronic computers after their invention with the american ENIAC, one of, if not the first machine to fit the modern definition of a computer, being used to calculate the hydrodynamics of shock waves for the development of the thermonuclear bomb, and another, the MANIAC being used to run the first Monte-Carlo simulation [Ste13]. Since then they have firmly established themselves in many areas, from physics and chemistry to material science, engineering and medicine [Bor21].
Their place right inbetween experiment and theory stems from their ability to recreate experimental results and predict the effects of assumptions made in new theories. Recreating empirical observations in a simulation allows for a dissection of the causalities and a better understanding of the recorded data. But they can also be used to run experiments that are not possible or desirable in real life, from things so small they are too difficult to measure, to astronomical processes happening on a timescale inconceivable to the human mind, or nuclear explosions and meteorite impacts, which while interesting as an object of study are generally considered undesirable to have occur on this planet. Another benefit over experiments is the lack of unknown external factors, such as contaminations in reactants, temperature fluctuations in a sample or inaccurate measurement devices. For theorists they can assist by showing the consequences of the laws that were derived from their newest assumptions in various scenarios. Molecular dynamics simulations are a special kind of simulation whose subject of interest are molecules and atoms. It is obvious that the simulated volumes must be very small, because of the extremely high particle density in most gases, liquids or even solids. Due to the large amount of interactions between those particles in any given timeframe, the time step must also be very small to remain accurate enough to the time-continuous model to be meaningful. Therefore the simulated timeframe has to be chosen very carefully for the computation to be able to finish in a reasonable amount of time. They were pioneered in the 1950s and see extensive use in all areas were phenomena are studied that are hard or impossible to model with continuous techniques [BZBP13, AW57]. The molecular dynamics simulations that are relevant for this work follow the laws of classical mechanics, most importantly Newton's second law which states that $\vec{F} = m\vec{a}$, with m being

the mass of the particle, $\vec{a}$ its acceleration and $\vec{F}$ the force acting on it. It is also assumed that all particle positions and velocities are known exactly at all times and the speed of force propagation is infinte, meaning forces act instantly. In theory, this makes the simulation inherently deterministic, although in practice the inaccuracies of floating point operations can introduce some unpredictable behavior [Ste13]. The consequence is, that the effects of quantum mechanics are disregarded.

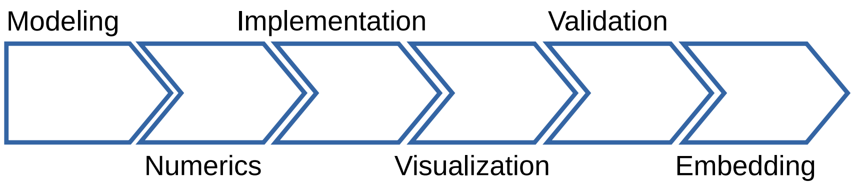


Figure 1.1.: The simulation pipeline

The process of running any kind or simulation is made up of several steps, a so called simulation pipeline (Figure 1.1).

Modeling Creating a formal, usually simplified description of a suitable excerpt of the object of interest.

Numerics Deriving equations from the model and suitable algorithms to solve these

Implementation Writing code on the target architecture that implements the developed algorithms in a time- and memory-efficient manner

Visualization Processing the gathered data to allow for interpretation of the result

Validation Searching for potential sources for errors in the model, algorithms, implementation and interpretation by comparing the results with those from other models, algorithms, implementations and experiments

Embedding Simulations are only meaningful in their context, for example a research and development campaign, and have to be integrated in their surrounding processes.

A downside of simulations that must be mentioned here, is that in each of the first three steps, inaccuracies are introduced that must be accounted for and measured if possible. The model is a simplification of the reality, therefore it will behave differently, at least in certain scenarios. The algorithms often discretize, usually by introducting time steps instead of treating time as a continuum, leading to inaccuracies whose magnitude is determined by the length of the chosed step as well as the ability of the algorithm to correct for them. Lastly, the implementation will introduce an error whenever floating point numbers are used, as they can only provide a limited accuracy and are subject to many different unwanted side-effects. This makes the validation step all the more important. The maximum theoretical error of an algorithm or implementation can be calculated but the best assessment of the effect on the end result is often given by a comparison with other data. This paper is concerned near exclusively with the implementation step. The model and accompanying equations have already been provided in the form of the inter-molecular potentials discussed the Theoretical Background chapter. As this work is a part of a research project to develop a framework for simulations there is also no need for validation and embedding, besides the measurement of inaccuracies caused by the new implementation and testing for its correctness. An simulation implementation can generally be divided into certain steps, which are shown in Figure 1.2.

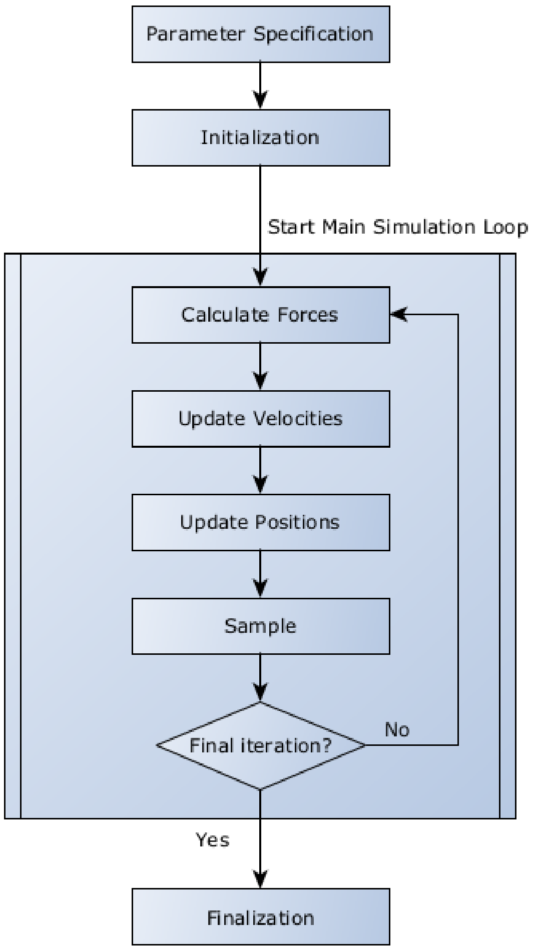


Figure 1.2.: General simulation program structure

Parameter Specification Set the conditions of the scenario such as initial temperature, number of particles, density or specific placement, time step, etc.

Initialization Initialize the system according to the previously set conditions, for example particle positions and velocities

Main Simulation Loop Start Gets run for each specified time step until the set end time is reached

Calculate Forces Compute the forces on all particles

Update velocities From the computed forces Newton's equations of motion are used to find each particle's new velocity

Update positions Combining the velocities and time step the particles' new positions are determined

Sample The metrics of interest are sampled. This can include particle positions, velocities, density or total global energy, among others

Main Simulation Loop End

Finalization From the gathered samples, averages or other interesting data points of the measured quantities are calculated and store, and any needed cleanup is performed

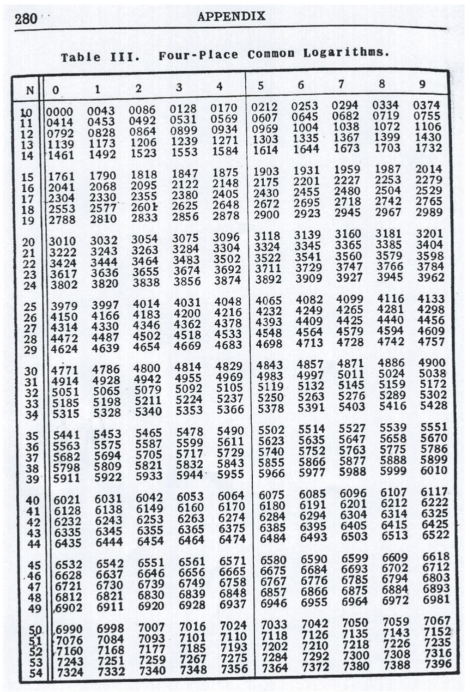
[FS23]

Obviously by far the most important part to optimize is everything in the main simulation loop, just based on the number of times it is executed ranging from thousands to millions and beyond. Out of the four steps in the loop, the computation of the particle forces reigns above all else in time-consumption due to the complex equations involved and the fact that if no other optimizations are performed to reduce them, the number of calculations scales with the square of the number of particles, or even the cube if a three-body potential is used [FS23]. To be exact, the computational complexity scales with $O(n^m)$ with n being the amount of total particles and m the number of particles partaking in a single interaction. The other steps need to be performed at most once per particle which makes them not irrelevant, but definetely less critical for the performance of the implementation as a whole. Because of that the optimizations discussed in this work are concerned exactly with this performance critical step.

2. Theoretical Background

2.1. Lookup Tables

The principle of a lookup table is as simple as it is old. If the result of an expensive to calculate function is required often, one can decide to precompute these results and save them in a table, where they can be looked up easily and quickly during the actual computation. Depending on the function and context, the table can either include the result for every possible input value, or a range of values selected according to some logic. Traditionally popular functions where lookup tables were applied were the trigonometric functions or logarithms, because exact computation of their values required many steps and lengthy algorithms such as the Taylor series. The concept of using a table of precomputed values to speed up other calculations dates back at least to ancient mathematicians of Sumer and circa 2600 BCE. Since then they have become increasingly popular throughout history [CKCFR03]. An example of such a manual lookup table for logarithms can be seen in Figure 2.1



280 APPENDIX

Table III. Four-Place Common Logarithms.

N	0	1	2	3	4	5	6	7	8	9
10	0000	0043	0086	0128	0170	0212	0253	0294	0334	0374
11	0414	0453	0492	0531	0569	0607	0645	0682	0719	0755
12	0792	0828	0864	0899	0934	0969	1004	1038	1072	1106
13	1139	1173	1206	1239	1271	1303	1335	1367	1399	1430
14	1461	1492	1523	1553	1584	1614	1644	1673	1703	1732
15	1761	1790	1818	1847	1875	1903	1931	1959	1987	2014
16	2041	2068	2095	2122	2148	2175	2201	2227	2253	2279
17	2304	2330	2355	2380	2405	2430	2455	2480	2504	2529
18	2553	2577	2601	2625	2648	2672	2695	2718	2742	2765
19	2788	2810	2833	2856	2878	2900	2923	2945	2967	2989
20	3010	3032	3054	3075	3096	3118	3139	3160	3181	3201
21	3222	3243	3263	3284	3304	3324	3345	3365	3385	3404
22	3424	3444	3464	3483	3502	3522	3541	3560	3579	3598
23	3617	3636	3655	3674	3692	3711	3729	3747	3766	3784
24	3802	3820	3838	3856	3874	3892	3909	3927	3945	3962
25	3979	3997	4014	4031	4048	4065	4082	4099	4116	4133
26	4150	4166	4183	4200	4216	4232	4249	4265	4281	4298
27	4314	4330	4346	4362	4378	4393	4409	4425	4440	4456
28	4472	4487	4502	4518	4533	4548	4564	4579	4594	4609
29	4624	4639	4654	4669	4683	4698	4713	4728	4742	4757
30	4771	4786	4800	4814	4829	4843	4857	4871	4886	4900
31	4914	4928	4942	4955	4969	4983	4997	5011	5024	5038
32	5051	5065	5079	5092	5105	5119	5132	5145	5159	5172
33	5185	5198	5211	5224	5237	5250	5263	5276	5289	5302
34	5315	5328	5340	5353	5366	5378	5391	5403	5416	5428
35	5441	5453	5465	5478	5490	5502	5514	5527	5539	5551
36	5563	5575	5587	5599	5611	5623	5635	5647	5658	5670
37	5682	5694	5705	5717	5729	5740	5752	5763	5775	5786
38	5798	5809	5821	5832	5843	5855	5866	5877	5888	5899
39	5911	5922	5933	5944	5955	5966	5977	5988	5999	6010
40	6021	6031	6042	6053	6064	6075	6085	6096	6107	6117
41	6128	6138	6149	6160	6170	6180	6191	6201	6212	6222
42	6232	6243	6253	6263	6274	6284	6294	6304	6314	6325
43	6335	6345	6355	6365	6375	6385	6395	6405	6415	6425
44	6435	6444	6454	6464	6474	6484	6493	6503	6513	6522
45	6532	6542	6551	6561	6571	6580	6590	6599	6609	6618
46	6628	6637	6646	6656	6665	6675	6684	6693	6702	6712
47	6721	6730	6739	6749	6758	6767	6776	6785	6794	6803
48	6812	6821	6830	6839	6848	6857	6866	6875	6884	6893
49	6902	6911	6920	6928	6937	6946	6955	6964	6972	6981
50	6990	6998	7007	7016	7024	7033	7042	7050	7059	7067
51	7076	7084	7093	7101	7110	7118	7126	7135	7143	7152
52	7160	7168	7177	7185	7193	7202	7210	7218	7226	7235
53	7243	7251	7259	7267	7275	7284	7292	7300	7308	7316
54	7324	7332	7340	7348	7356	7364	7372	7380	7388	7396

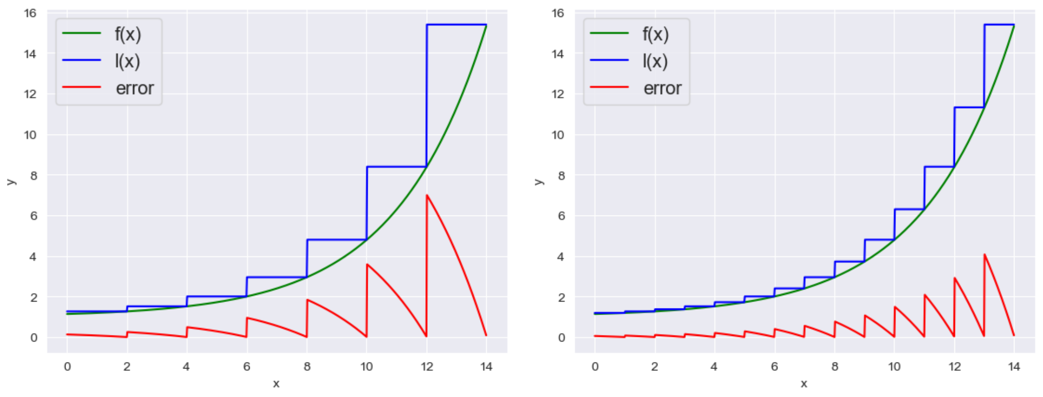
Figure 2.1.: Historic logarithm lookup table [Qui]

Lookup tables already became a topic in computer science at the infancy of computers. In the beginning one of the primary purposes of these machines was the calculation of such

tables, which were previously calculated by hand. For finding tables to solutions to problems which could not easily be represented in a single formula, simulations were employed. Very famously in 1954 a group of researchers successfully calculated the best strategy for every possible combination of cards in the game of blackjack by running millions of simulated hands on an IBM 701 [IBM11].
Since then, they have become a staple of program optimization both in regular software but also in hardware [Fog24, HLR+23, Mav15, DLR+15, SB94]. They can be used to speed up the evaluation of a function, if it is expensive enough to compute so retrieving its value from memory is faster than the computation. Then, by precomputing the result of the function over a domain of inputs, expensive runtime evaluations can be replaced with cheaper table lookups leading to significant performance gains [PF05]. The issue of lookup latency has become more and more significant, the larger the gap between the processing speed of chips and the access time of memory became. Therefore tremendous care must be applied to ensure the table remains in a memory level, be it L1, L2, or L3 cache or even main memory, where the access times are lower than the calculation time. This usually means that compromises in other areas of the table design must be made to ensure the table size remains small enough to fit in the desired memory location.
Most generally a lookup table can be classified according to two criteria, first being dimensionality. This referres to the number of parameters required to compute a result's table index and is usually corresponding to the number of parameters required by the precomputed function [PF05]. The second important property is the granularity. If the evalutated points are spread uniformly over the function domain the granularity is defined as

$$Granularity = \frac{Domain}{Size} \tag{2.1}$$

with $Size$ referring to the number of table entries [Wil12]. In theory the best lookup table would allocate 2^u entries, each v bits wide, with u being the combined length of the function parameters in bits and v the size of the output. This works for very small input and entry sizes, but with today's standard number representations often being 32 and 64 bit long this naive implementation is rarely practical or efficient [Par19]. Obviously the number of entries need to be limited somehow. A popular option is to allow for some pre- or post-computation of the table entry. This can be useful when simple transforms of either the input operands or the result can yield great reductions to the table size, for example through the use of symmetry present in the function. Such optimzations are dependent on the function itself and there are tradeoffs to be considered between computation time and lookup table size [Par19].
For non-trivial functions this is often not possible, or just not enough to permit a 1-to-1 mapping, where for every possible inputs its exact function result is stored. Therefore another compromise has to be struck, this time between size and accuracy. Instead of mapping every input value to its exact function result, an intervall is chosen and only one point per intervall is computed and stored. On lookup the intervall containing the desired input is determined and then the result assigned to this range returned. Depending on the specific funtion and interval size this can add a significant error.



(a) Error from lookup with 7 intervals (b) Error from lookup with 14 intervals

Figure 2.2.: Error of lookuptables with different interval sizes

It is immediately obvious that the error increases with the rate of change in the intervall. A possible optimization preserving both accuracy and table size could therefore be, to divide the domain into multiple subranges and implement a different lookup scheme with differing granularity for each. Notably this does make the process of determining the correct index in the corresponding subtable more complex, offering yet another trade-off that has to be considered and evaluated [Par19].
Another possible solution to the issue of keeping the table size relatively small while retaining a high accurary is the use of an interpolation function. They aim to, instead of directly storing more evaluations in the table, approximate the function output for inputs between the stored ones. The most basic one is what was already discussed, selecting the nearest value for every input, which is called `direct access`, `nearest-neighbor` or in this work `next neighbor` interpolation [Wil12, PF05]. More complex and accurate interpolation algorithms combine multiple entries representing function evaluations near the desired point. A comparatively simple scheme that still provides a significantly error reduction is the `linear interpolation`. Here two entries are retrieved, corresponding to the two neighbors of the desired result and then combined to an average weighted according to their respective distance from the point[Wil12]. This is done according to the formula

$$y = y_1 + (x - x_1) * \frac{(y_2 - y_1)}{(x_2 - x_1)} \tag{2.2}$$

where (x_1, y_2) is the value of the evalutated function at x_1 and (x_2, y_2) at x_2 respectively and x_1 and x_2 are points stored in the lookup table. x_1 is the smaller of the two.

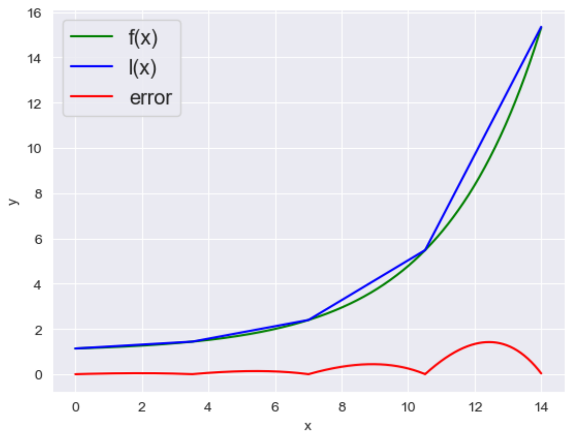


Figure 2.3.: Error from lookup with linear interpolation and 4 intervals

The advantage in error-reduction is immediately obvious if Figure 2.3, where with only 5 stored points an accuracy similar to the next neighbor with 14 points can be achieve. An additional point needs to be stored to provide a partner for interpolation in the lowest interval. The obvious disadvantage is that a second table lookup is needed and additional computation needs to be done after the memory access. There exist many more interpolation algorithms requiring more table entries to be retrieved or more calculations to be performed on the results, which can achieve even better accuracy, making for yet another trade-off to be considered when choosing the correct lookup table implementation strategy [Par19].
Yet another optimization option would be to only store a part of the domain in a table, and compute the excess during runtime. Usually the stored values will be those, that are evaluated the most frequently. What those values are, can be determined either during the implementation, if enough knowledge about the runtime scenaio exists, or determined during runtime in which case the table would behave more like a cache rather than what is traditionally considered a lookup table. Either implementation does however present the need for additional conditional logic to check if the desired value is present in the table or needs to be computed, and conditionals are expensive [Wil12]. An additional possibility, which will play a role in the implementation of the Lennard-Jones lookup table, is for the table to serve the values in the subdomain in which the function changes very little, keeping the resulting error small, and calculating the values for the more critical region during runtime.

2.2. Lennard-Jones Potential

The Lennard-Jones potential is an intermolecular pair potential that describes the interactions between a pair of uncharged particles. It was first described by British mathematician and professor of theoretical physics Sir John Edward Lennard-Jones in 1931. It has been relevant in computer based modeling since the beginning of scientific computing and has remained the most popular and most thouroughly studied intermolecular potential ever since [LSH24].
The Lennard-Jones potential is defined as

$$V_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.3}$$

where r is the distance between the two particles ϵ is the depth of the potential well and σ is the distance at which the potential energy between the two particles reaches 0. These are obviously dependent on the types of interacting particles [FW23]. The term is comprised of two simple parts, modeling a repulsive and an attractive force component as seen in Figure 2.4. A repulsive 12-term models the Pauli repulsion of atoms which are very close together. Its exponent 12 was chosen not based on any scientific theory, but because it fit the empirical data. The exponent 6 in the attractive term is derived from the strength of the van der Waals forces [FW23].

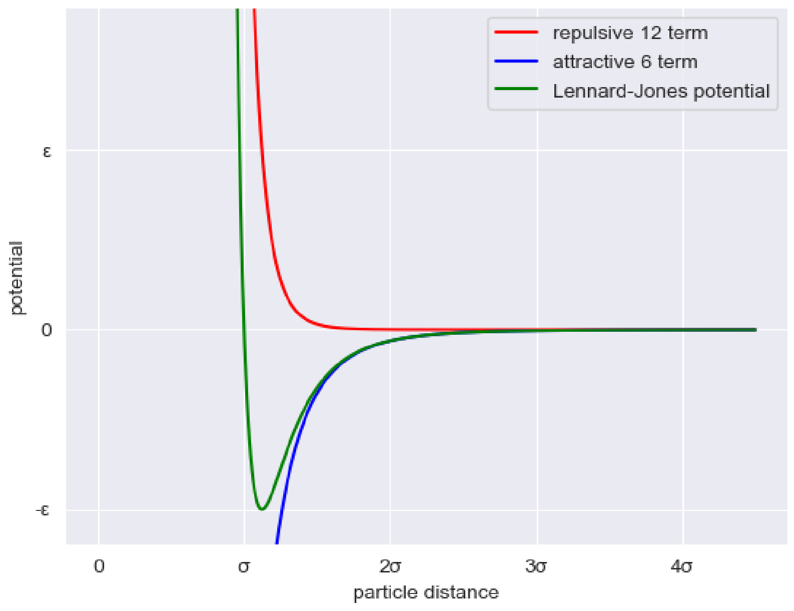


Figure 2.4.: Lennard-Jones potential and its two parts

While mathematically particles affect each other over an arbitrarily large distance, in computing to save on processing power and increase simulation speed, one often employs a cutoff distance, after which the force between the particles is assumed to be 0. This is possible due to the rapid convergence towards 0 making the Lennard-Jones a so-called short-range potential, which allows the employment of a cutoff without a significant divergence from the fully calculated simulation.

2.3. Axilrod-Teller Potential

The Axilrod-Teller potential is most often used in combination with a pair-wise potential such as the aforementioned Lennard-Jones potential [Att92, Sad98]. It is a three-body potential providing a correction to the modeling of the van der Waals interactions and was introduced by B. M. Axilrod and E. Teller in 1943 [AT43]. They derived it by applying third-order Rayleigh-Schrödinger pertubation theory to a triplet of noble-gas atoms [PBJB84]. This straightforward process gives a formula of

$$E_{ijk} = Z\frac{1 + 3\cos\theta_i\cos\theta_j\cos\theta_k}{(r_{ij}r_{jk}r_{ki})^3} \tag{2.4}$$

to calculate the interaction energy between the particle i, j and k. r_{ij} represents the distance between particles i and j, and θ_i the angle at point i between the lines i-j and i-k. Z is the so-called the Axilrod-Teller coefficient and is the only parameter in the equation [BC21]. In the original paper by Axilrod and Teller Z is described as a positive quantity of the order $V\alpha^3$. V is the ionization energy and α is the polarizability of the interacting particles [AT43]. Obviously the three-particle interaction results in a lot more combinations resulting from the same amount of particles than with a two-particle potential, which requires many more calculations. Also notable is that the potential can result both in attracting, but also repelling forces. This is due to the three cosines, which depending on the angles can result in a negative numerator and as such a negative result as the distances and therefore the denominator, as well as the Axilrod-Teller coefficient Z are always positive. Specifically the term is positive (E_{ijk} repulsive) if all three angles are $< 90°$ and negative (E_{ijk} attractive) if one angle is $> 90°$, reaching its minimum of -2 when all three particles are on a straight line [PBJB84]. For better understanding of the term's behavior, Figure 2.5 is provided.

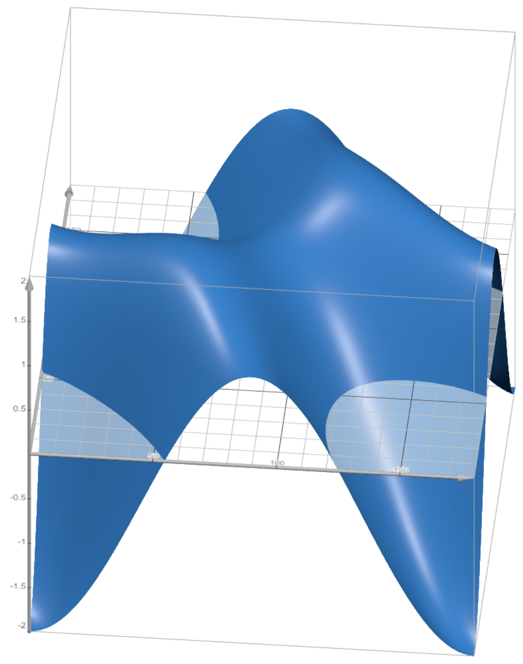


Figure 2.5.: 3D graph of the Axilrod-Teller numerator with θ_i and θ_j as the axis, running from 0° to 180°. θ_k is determined by the sum of angles in a triangle.

From observation of the term it is obvious, that the effect of the Axilrod-Teller interaction increases greatly, the closer the particles are to each other.

2.4. AutoPas

AutoPas is an open source C++ library for particle simulations in active development primarily at the Technical University of Munich. It specializes in simulating N-body systems governed by short-range interactions. During the development of such a simulation the researcher or developer must make many important decisions pertaining to the algorithms used, data layout, parallelization strategies and other optimizations. The most favourable combination of choices in each of these areas depends greatly on the exact scenarion being simulated. Therefore this selection process requires extensive background knowledge and experience. Additionally over the course of the running simulation this optimal selection can change, opening up the possibility of great performance losses even if the initial setup was chosen with care [GSBN22].
AutoPas seeks to provide a solution to this complex problem through a process termed automated algorithm selection, a type of auto-tuning. During the simulation run it measures the speed of different configurations and uses the sampled data to decide the optimal selection for a user-specified amount of future iterations [GST+19]. There are currently four different parameters that can be adjusted by AutoPas. The first optional optimization is the use of Newton's third law of motion. This states, that for every force exerted on a body i by a body j, a reactive force, equal in magnitude and opposite in direction is exerted on body j. In practice this amounts to a near 50% reduction in the number of calculations needed, as every force on a particle i interacting with a particle j can be very easily and inexpensively be transformed into the force acting on particle j during the interaction with particle i. In some cases, for example in the simulation of Smooth Hydrodynamics, the pair-wise potentials used are not symmetric, so the Newton3 optimization isn't possble [GSBN22].

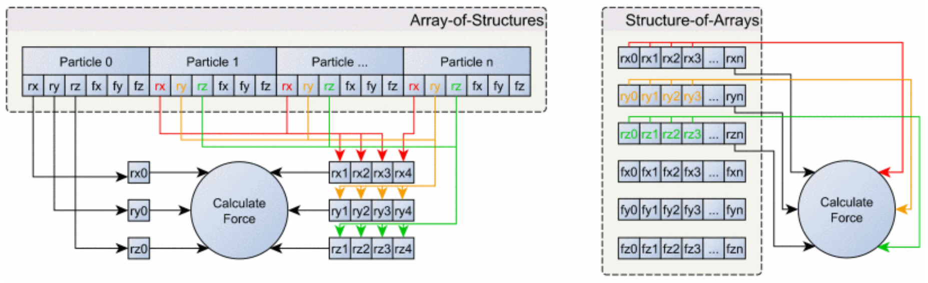


Figure 2.6.: AoS and SoA Container data retrieval [GST+19]

One of the most important features of any particle simulation implementation is the layout in which the particles are stored. As seen in Figure 2.6 storing the particles in a structure of arrays instead of an array of structure allows data to be loaded into vector registers more easily, as they can be accessed in continuous batches, greatly increasing the throughput when utilizing SIMD-instructions. The array of structures layout has the advantage of easier access to single particles with all their information, which can make it faster to transfer them between MPI ranks [GSBN22].
Because AutoPas focuses on short-range interactions, it implements a cutoff-distance after which the force resulting from the influence of one particle on the other is considered to be equal to 0 and therefore doesn't need to be calculated. This is an optimizations which is

crucial for simulations with a very large number of particles to be possible at all. Due to the nature of the short-range potentials which are simulated, this does not introduce any significant error, but allows the computational complexity of the N-body problem to be reduced from $O(N^2)$ to $O(N)$ because the number of other particles in the cutoff radius is independent of the total number of atoms in the simulation scenario [GSBN22, EP10]. With some additional correction calculations this can even be done correcting Therefore another important parameter is the data structure in which the particles are stored, which in turn determines the method used for finding which pairs can acutally interact with each other. Doing this efficiently is required to gain any performance from this optimization. For this reason AutoPas implements many different particle containers, each with different use-cases [GST+19].
The last important parameter AutoPas can dynamically adjust is the parallelization strategy. For multithreading the popular OpenMP[1] API is used. The two metrics that must be optimized for here are minimal scheduling overhead combined with a maximized spread of load among the available processor cores. If the exact simulation scenario is not known beforehand, these two goals are in conflict with each other. Also, if the Newton 3 optimization is enabled, race conditions can occur if a processor tries to access the second particle but it is owned by another, which must be accounted for and can result in additional overhead [GST+19]. All of these parameters can be adjusted by AutoPas, although the user is left the choice of restricting the allowed options. Importantly this should only be done if assumptions about the simulated scenario can be made, that allow for reasonable restrictions on the algorithm choices. All of this enables domain scientists without extensive expertise in the implementation of N-body particle simulators to write code which will dynamilcally optimize itself to deliver faster simulation speeds than would otherwise be possible. To implement AutoPas in their own simulator, the user has to provide two classes. A custom particle class which contains all properties and information the simulated particle should possess. To ensure compatability, the class must be derived from the provided `autopas::ParticleBase` class. The second requirement is the pairwise force calculation that shall be applied to each pair or triplet of particles in every iteration. The classes implementing these are called *funtors* and at least one is required. They too have to be derived from template classes provided by AutoPas, either `autopas::PairwiseFunctor` or `autopas::TriwiseFunctor` which are themselves derived from `autopas::Functor`. The functor, or functors can use any information saved in the particle class for their calculations. If the interactions between different kinds of particles have to be modeled, a type-ID flag can be implemented in the particle class to allow the functor to distinguish the differnent types of particles [GSBN22]. In the functor compile-time optimizations for example the use of SIMD-extensions implemented in the users processor such as SSE and AVX in the x86-64 architecture, can be implemented by the user. Also possible would be the selection of different algorithms based on some parameter during the compilation. These and similar optimizations are referred to as static tuning, compared to the dynamic tuning which is AutoPas' main appeal [GST+19].
The author's work was done in an molecular dynamics simulator called md-flexible that is shipped as an example implementation with AutoPas. The Lennard-Jones and Axilrod-Teller Functors implemented in the md-flexible simulator, in which the lookup tables were imple-

[1]https://www.openmp.org/

mented, are derived from the `autopas::PairwiseFunctor` and `autopas::TriwiseFunctor` respectively.

Part II.

Implementation

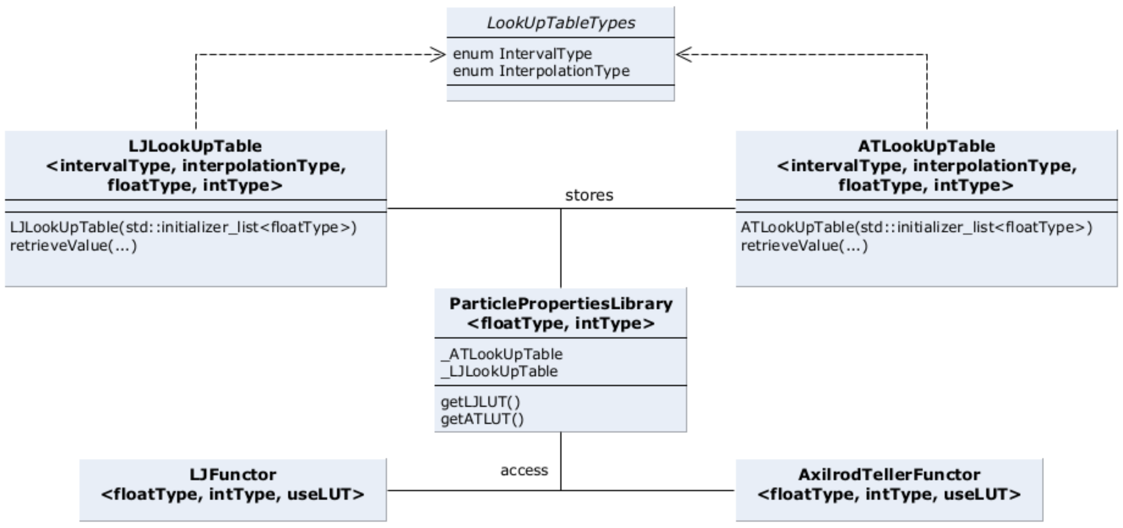

LookUpTableTypes
enum IntervalType
enum InterpolationType
LJLookUpTable <intervalType, interpolationType, floatType, intType>
LJLookUpTable(std::initializer_list<floatType>)
retrieveValue(...)
ATLookUpTable <intervalType, interpolationType, floatType, intType>
ATLookUpTable(std::initializer_list<floatType>)
retrieveValue(...)
stores
ParticlePropertiesLibrary <floatType, intType>
_ATLookUpTable
_LJLookUpTable
getLJLUT()
getATLUT()
access
LJFunctor <floatType, intType, useLUT>
AxilrodTellerFunctor <floatType, intType, useLUT>

Figure 2.7.: Implementation Structure Diagram

All implementation work was performed in md-flexible, a programm integrating AutoPas in a working molecular dynamics simulator to show off its features as well as serve as a guide for new users seeking to use the library in their own work. In the repository the relevant files can be found under `AutoPas/applicationLibrary/molecularDynamics/molecularDynamicsLibrary/`. In total three important files were added and another three had important modifications done to them to implement the lookup scheme. A new namespace `ForceLookupTable` was introduced to bundle all new classes and enumerations and prevent pollution of the global namespace. Following the general style of md-flexible functors, namely the `LJFunctor` and `AxilrodTellerFunctor` which the lookup tables enhance, the tables are implemented as templated classes. This has the advantage of removing unneccessary conditionals from the retrieveValue method which could slow down execution at runtime. Because the performance of the table lookup is absolutely critical for the performance of the entire simulator, the extra compilation time caused by the extensive use of templates is accepted.
Besides inheriting the `floatType` and `intType` parameters from their respective functors for compatiblility, they both expect two more parameters. The first one is `intervalType`, which is currently unused. It is intended for potential future options to control the placement of evaluation points. As both the Lennard-Jones and Axilrod-Teller potential graphs' slopes are significantly steeper the closer the particles are to each other and nearly flat once the distances are great enough, it could be beneficial to place more points in the first part and less in the second, as this would decrease the maximum error from accessing a value between the points, while keeping the table size small, at the cost of a more complicated access function.
The second parameter, `interpolationType` designates the interpolation function which should be used when retrieving values from the table. Such a function was implemented `LJLookUpTable` and will be discussed in Figure 3.2.
The two template parameter are each implemented as a custom enumeration, called `IntervalType` and `InterpolationType` respectively, that allows a convenient overview of the available options and prevents any need for parsing logic which would be required for

strings, while keeping the obvious names, removing any potential for user error caused by ambiguous names.
The tables themselves are stored in the `ParticlePropertiesLibrary`, which also stores the physical properties of the different molecule types, and therefore provides all information needed for their generation. While this requires the table to be retrieved from the `ParticlePropertiesLibrary` on every lookup, due to the implementation of AutoPas the force functors get rebuilt on every iteration, which would cause the tables to be completely rebuilt as well. The `ParticlePropertiesLibrary` on the other hand does not have this limitation, so the tables only have to be constructed once, during initialization. Technically the `floatType` and `intType` parameters are also determined from here, but as they should be the same everywhere in the code, this hardly makes a difference.
The `LJFunctor` and `AxilrodTellerFunctor` class templates were expanded with an additional `useLUT` parameter which is used in combination with a `if constexpr`-statement in the calculation to make the decision about accessing the lookup table or performing the force calculation at runtime, can be at compile time, preventing slow branches at runtime.
The inner workings of both tables are very similar. They feature constructors taking a `std::initializer_list<floatType>` as an argument. This is neccessary because due to the (potentially) many possible combinations of interval and interpolation type many different parameters could be required, in-turn causing massive amounts of code duplication. The downside two-fold. Firstly, not all parameters are floating point numbers, requiring casting, and secondly the great potential for user-error due to mistakes in the ordering of input arguments. Mitigation was attempted by thouroughly documenting the argument-structure and most importantly ensuring that the user never has to directly interact with the constructors. Depending on the `intervalType` a fill function is called by the constructor that initializes the table with values according to the selected point distribution startegy.
The second important function visible to the outside is the `retrieveValue` method. This is what is called in the force functors to access the lookup table and provides a uniform interface independent of the choices in interval and interpolation, decoupling the lookup table strategy from the functor implementation and improving the developer experience for the user. Their parameters are therefore only dependent on the force functor and not the internals of the table. Internally `retrieveValue` contains a number of `if constexpr`-statements to determine how the desired value should be retrieved without any additional runtime branching. From here the appropriate interpolation function is called, as only there can it be determined which and how many elements have to be read from the lookup table. The interpolation will then, again using `if constexpr`-statements, check the `intervalType` and call the appropriate `getIndex` method to compute the location of the required entries in the table. From there it will interpolate and return the result to the force functor.
Internally the entries are stored in a `std::vector` due to its flexibility in ordering the entries as well as most importantly its superior random access speed. For filling the vector, both classes also include a copy of their respective force functors.

3. Lennard-Jones Lookup Table

The Lennard-Jones lookup table is a one-dimensional lookup table as the only parameter that is unknown at runtime is the distance between the two particles whose forces need to be computed. All other paramters neccessary for the calculation of the Lennar-Jones potential, sigma and epsilon, as well as the cutoff distance, are provided to the table in its constructor and stored internally. Its `retrieveValue` method takes as its only argument the squared distance between the two particles. This is calculated in the `LJFunctor` prior to the call to the lookup table by means of a dot product of the distance vector between the two particles with itself. Mathematically the formula looks like $(v1 - w1)^2 + (v2 - w2)^2 + (v3 - w3)^2$ with vX and wX representing the position coordinate of particle v and w respectively in dimension X. In an effort to achieve good accuracy without an overly large table size, different interpolation methods and other improvements were tested which will be discussed in the following.

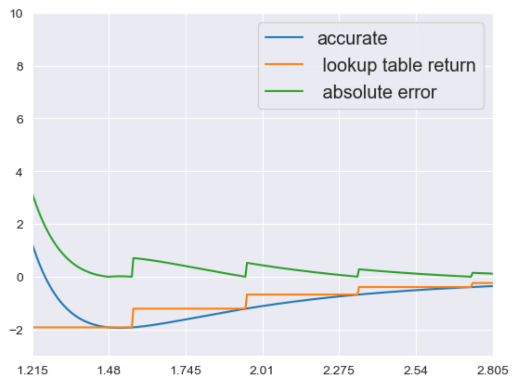
3.1. Next Neighbor

```
floatType getNextNeighbor(floatType dr2) {
  if (dr2 >= cutoffSquared)
    return lut.at(numberOfPoints-1);
  else
    return lut.at(std::floor(dr2 / pointDistance));
}
```

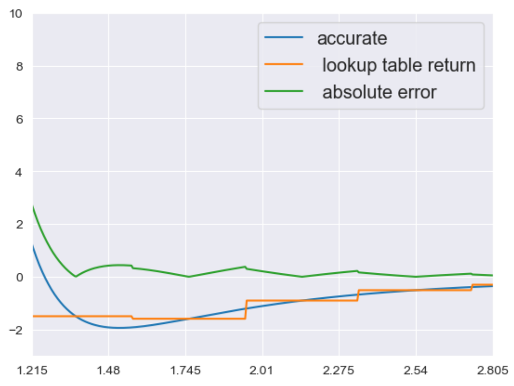
Listing 3.1: Simplified code for the next neighbor table lookup

The first interpolation function that is implemented is a naive next neighbor approach. Here the input `dr2` is divided by the distance between the reference points in the lookup table. Flooring the result of this computation gives the interval in which the point is located. As a special case, if `dr2` is exactly the `cutoffSquared`, this computation would result in an off-by-one error and therefore an out of bounds access. There should never be a value greater than `cutoffSquared` as they should have been filtered out by the force functor, but for safety it is left in the code.
For example, a `cutoffSquared` of 2, combined with a `numberOfPoints` value of 4 would result in 4 distinct intervalls each with a size of $\frac{2}{4} = 0.5$ which is at the same time the `pointDistance` between the reference points which get stored in the lookup table. If there is now a table access with a `distanceSquared` between the two particles of 0.75, the `retrieveValue` and from there the `getNextNeighbor` method is called with `dr2` set to 0.75. 0.75 is obviously not equal or greater than the `cutoffSquared`, so we move to the `else` part of the code. $\frac{0.75}{0.5} = 1.5$ and $\lfloor 1.5 \rfloor = 1$, placing us in the second interval, as array-indexing starts at 0. The second interval goes from $1 * pointDistance = 0.5 \leq 0.75$ to $2 * pointDistance = 1 > 0.75$ and is therefore the correct interval. So the return value is

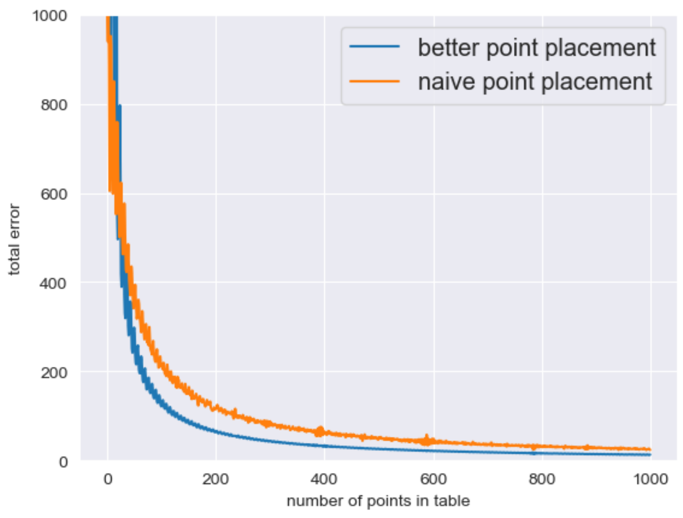
the point in the lookup table which represents all values in this interval.



(a) Return values from the lookup table when using the naive point placement graphed against the actual Lennard-Jones potential and the absolute difference between the two.



(b) Return values from the lookup table when using the better point placement graphed against the actual Lennard-Jones potential and the absolute difference between the two.



(c) The total error summed up over 1000 measurements from distances 0.95 to 6.25 measured on both kinds of point placement with increasing table size

Figure 3.1.: Comparison of the naive and better point placements

As seen in Figure 3.1a selecting the point at the beginning of the interval as representative for the whole introduces an unneccessarily big error. A simple optimization that can be performed is to instead select the point in the middle of the diagram, as done in Figure 3.1b. This reduces the total error as seen in Figure 3.1c. Notably, this optimization can be implemented without any additional cost, except a slightly more complicated table fill function. Instead of

```
void fillTableEvenSpacing () {
  for (auto i = 0; i<numberOfPoints; i++) {
    lut.push_back(LJFunctor((i+1) * pointDistance));
  }
}
```

Listing 3.2: Naive point placement for filling the table with evenly spaced points

it now has to be

```
void fillTableEvenSpacing () {
  for (auto i = 0; i<numberOfPoints; i++) {
    lut.push_back(LJFunctor((pointDistance/2) + (i * pointDistance)));
  }
}
```

Listing 3.3: Improved point placement by placing reference points in the middle of their respective intervals

but everything about the lookup logic itself stays the same.

3.2. Linear Interpolation

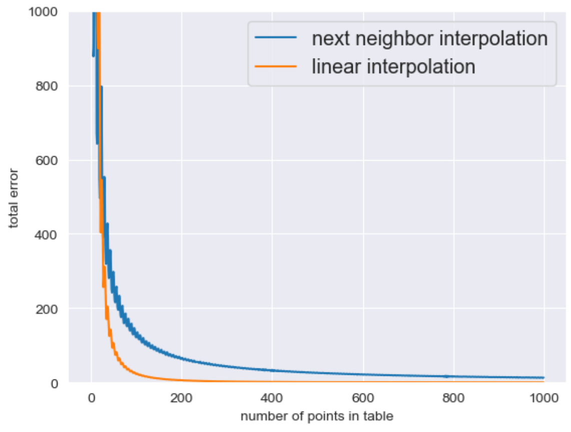


Figure 3.2.: Comparison of the error from next neighbor and linear interpolation over different table sizes

As discussed in Section 2.1 a method to decrease the error caused by the lookup while keeping the table size is the employment of a interpolation function. Therefore a linear interpolation function was impemented in the Lennard-Jones lookup table. As can be seen in Figure 3.2 this does provide a sizable accuracy improvement over the next neighbor interpolation and therefore allows for a smaller table. Notable is also the far more rapid decrease in the total error which was measured exactly as in Figure 3.1c. The other important requirement for the viability of this approach over the next neighbor is that of its performance characteristics, which will be investigated in Section 6.1. The interpolation function is a direct translation of the formula shown in Section 2.1. To prevent divisions by 0, which can occur if the particle distance falls exactly onto a stored point, although unlikely due to the nature of floating-point numbers, this is checked. If such a case happens, naturally the stored point being the exact value is returned. If a distance short of the lowest stored distance is requested, the lowest stored point is returned, as there is no partner which would be required for the interpolation.

3.3. Optimizations

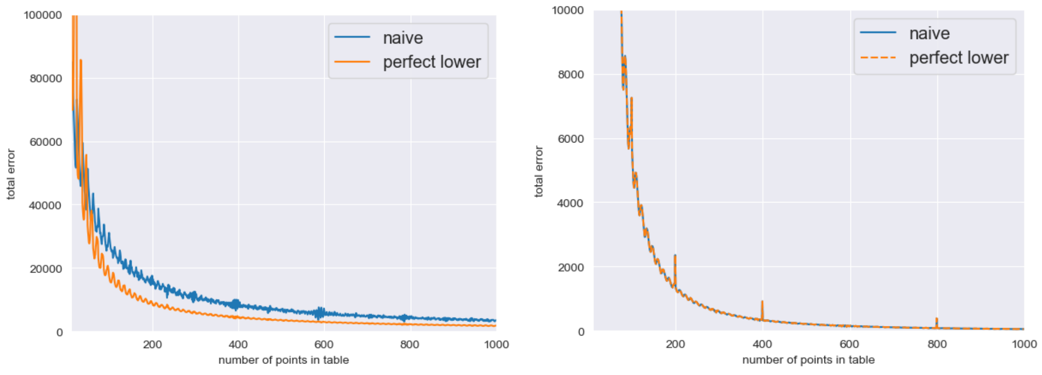


Figure 3.3.: Evaluation of the "perfect lower" optimization

As evident in Figure 2.4 the steepest slope in the graph is encountered close to particle distance 0. And as discussed in Section 2.1 the error caused by the use of a lookup table increases drastically with the slope of the function it is modeled after. A potential remedy for this was suggested in reducing the interval between stored points in steep-slope parts of the graph. In an effort to reduce the total error greatly without having to increase the table size, or lose out on the speedup from the use of a simpler version of this idea was tried out. If the distance to be computed is lower than the lowest stored in the table, instead of accepting the large error from returning this entry, the value is calculated accurately using the regular Lennard-Jones functor and then returned. This was indeed successful in reducing the error, both for the next neighbor interpolation as well as the linear interpolation implementation, as can be seen in Figure 3.3. Again, the performance implications are discussed in Section 6.1, although they should obviously be somewhere between those of the lookup table and regular functor implementations. Even the lower bounds check conditional was already required beforhand in the linear interpolation function, removing the only new potential for additional latency in this implementation.

4. Axilrod-Teller Lookup Table

4.1. Naive Implementation

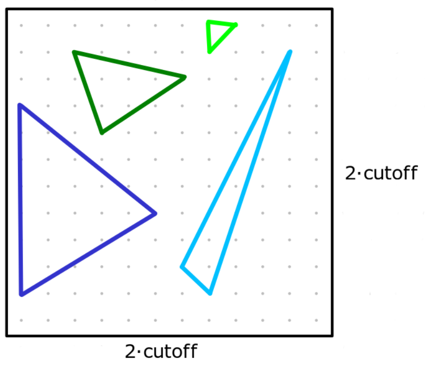
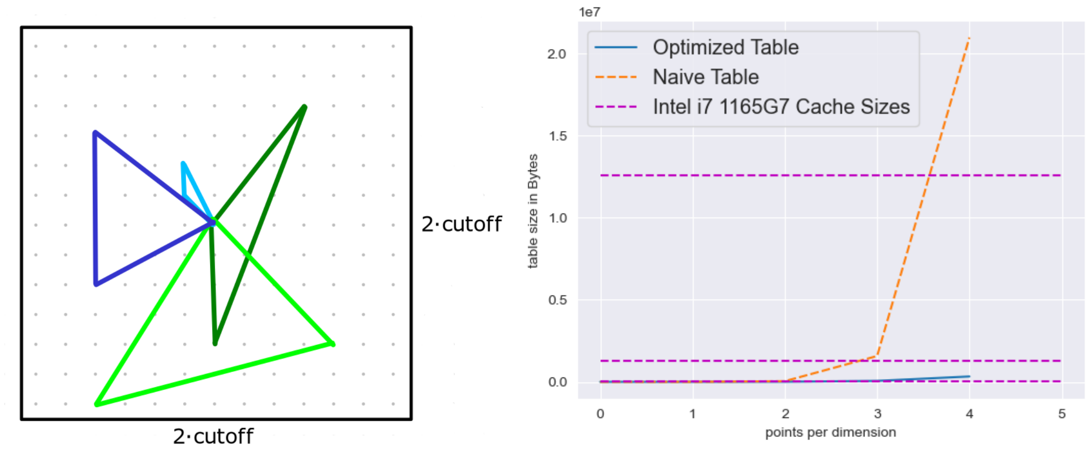


Figure 4.1.: Naive Axilrod-Teller Lookup Table Implementation which stores all possible relative particle positions

The Axilrod-Teller potential presents a much bigger challenge for efficient lookup table implementation as not one, but nine different parameters are required for its calculation. This being the vectors describing the relative positions of the three participating particles to each other in three-dimensional space, making for three vectors with three dimensions each. Each of these nine parameters has a range of $-cutoff$ to $+cutoff$ as this is the maximum distance two particles can be from each other. In reality obvioulsy only certain combinations of those nine parameters are possible, as the particles can only be $cutoff$-distance apart from one another in total, so an displacement vector of $(cutoff, x, y)$ is impossible for any $x, y > 0$. A naive approach to this would be to place points evenly spaced in the cubical volume defined by the corners $(-cutoff, -cutoff, -cutoff)$ and $(+cutoff, +cutoff, +cutoff)$ and store the return value of the Axilrod-Teller potential for each of those configurations. This approach is illustrated in Figure 4.1. Then at runtime the three input parameters are mapped to their three closest stored points and the stored value for these three points is returned. Other interpolation functions than this next neighbor approach would be possible, although the three-dimensional nature of the problem space makes them far more complex and require even more values than in the one dimension of the Lennard-Jones lookup table. Also important is that for a trio of small enough distances, they could all be mapped to the same point, asking the lookup table for the Axilrod-Teller potential of three particles in exactly the same location, which is undefined. So this must also be accounted for and prevented. The second major drawback is the space requirement of such a table. As there are nine parameters the number of points stored in the table increases with $O(n^9)$ with n being the number of stored points for each dimension.

4.1.1. Optimization to Exponent of 6



(a) (b)

Figure 4.2.: Optimized Axilrod-Teller Lookup Table Implementation which stores all possible relative particle positions where one particle is at the origin

A possible optimization for this problem is to map the first vector to the origin by subtracting its three components from their counterparts in the other two vectors. The effect of this idea is illustrated in Figure 4.2a. This removes all duplicate triangles and can be considered a requirement to make this approach at all usable in a real simulation. The massive effect of this optimization on the space-requirements of the table with increasing number of points can be seen in Figure 4.2b, although the difference is so stark that the line for the optimized table can barely be made out at the bottom, while that of the naive approach is already exiting the view after 5 points per dimension. There are however still duplicates being stored, in the form of triangles of the same shape, but differing rotation. An algorithm to remove those as well will be investigated in the next chapter.

4.2. Rotating Triangles

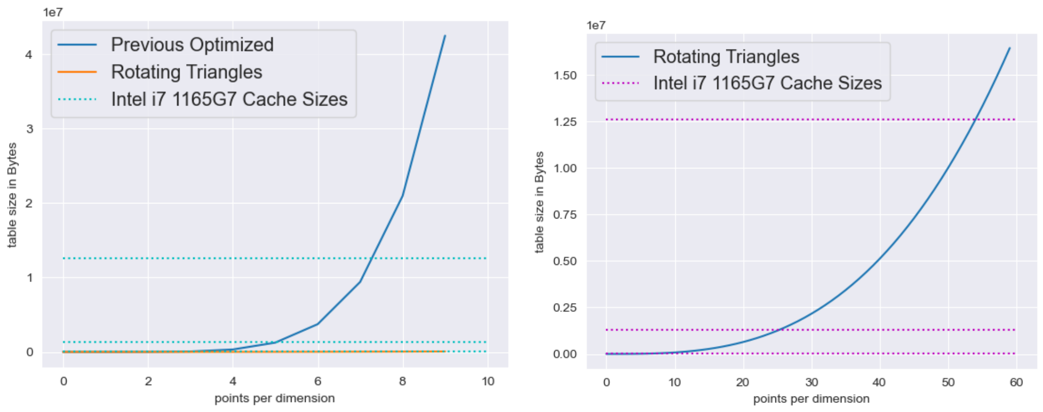


Figure 4.3.: Size growth of lookup table storing particles according to Figure 4.2a vs only the triangles

In theory, the lengths and relative directions of the force vectors are only dependent on the shape and size of the triangle defined by the displacements between the three particles. The only difference from there on is the rotation of this triangle in three dimensional space. The size and shape are also only determined by the three sidelengths. Therefore it is possible to implement a lookup table with only three parameters. This can again lead to large space-savings, as the table size now scales with $O(n^3)$ with n being the number of different sidelengths that shall be stored. This improves the table's growth rate and therefore the maximum resolution which can be used before higher-latency caches or even the main memory must be utilized for storage as seen in Figure 4.3. The major disadvantage of this implementation are the expensive operations required to find the rotation required to match the stored triangle to that made of the input particles and perform the rotation. This has major implications as discussed in Section 6.2.

4.2.1. Optimizations

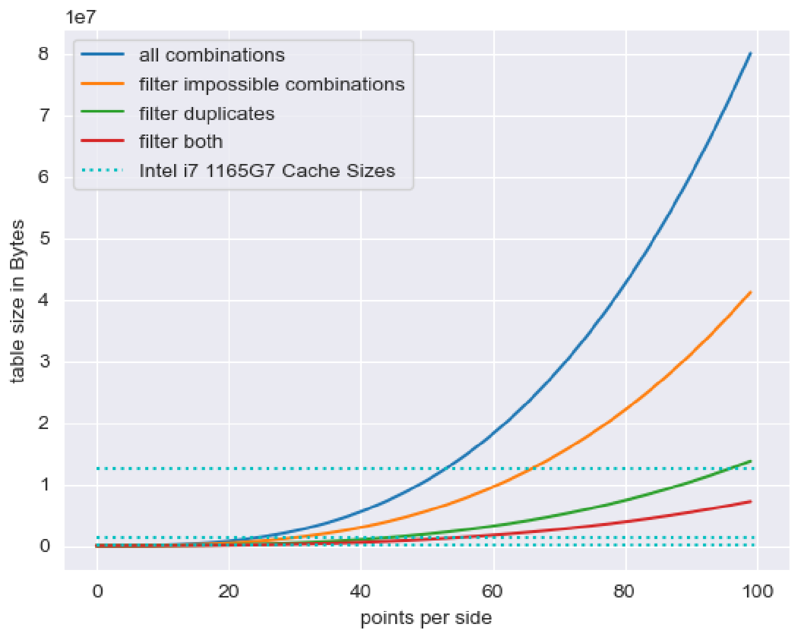


Figure 4.4.: Comparison of space saving measures in the triangle implementation

With this approach however, there is still potential for improvement in space on two fronts. First of all there are still duplicates stored, for example the triangles with sidelengths $(1, 2, 3)$ and $(3, 1, 2)$ are just rotated versions of the same triangle. Also the triangle defined by $(3, 2, 1)$ is the same shape, but the order of sides has been changed. Secondly some combinations of sidelengths don't make up a triangle, which can easily be checked using the triangle inequality $i + j >= k$. The question is which of these causes more wasted space. As can easily be seen in Figure 4.4 it is the amount of duplicated triangles. Therefore these were eliminated in this implementation by ensuring that the sidelengths of the stored triangles fulfill the inequality $i >= j >= k$.

5. Related Work

A very similar task as is this work was completed and implemented by [Str23]. They used a similar, lookup table based approach for the calculation of a combination of not only the Axilrod-Teller but also other contributing inter-particle potentials. Similarly to the rotating triangles optimization discussed here, the table consisted of triangles of different shapes and sizes which were then stored. What is very notable is that they, instead of generating the configurations by their three sidelengths, used two sidelengths and the encompassed angle. Their goal was to prevent storing any sidelength combinations which do not make up a viable triangle. As discussed in Section 4.2 this does happen in this work's implementation and causes a decent amount of wasted memory space. They however undertook no steps to exclude duplicate triangle configurations which as discussed in Section 4.2 is the far more important space-saving optimization. It does however make for a simpler lookup function as the complicated index calculation is not needed anymore. Instead however, the angle needs to be computed. Additionally they used a cubic interpolation function, which could have allowed them to compensate for the smaller table. Their paper does not include any information about the accuracy nor speed of their lookup table implementation, preventing any more detailed comparisons.

6. Benchmarks

6.1. Lennard-Jones

All measurements were performed on a Lenovo ThinkPad X1 Yoga G6 featuring an Intel i7 1165G7 processor and 32GB of memory. Available are 48kiB of L1d and 1.25MiB L2 cache per core as well as 12MiB of shared L3 cache. Installed on the system is Linux version 6.7.9-arch1-1 x86_64 and gcc version 13.2.1. For the Lennard-Jones benchmarks the simulation scenario found in `AutoPas/examples/md-flexible/input/fallingDrop.yaml` running for 2000 iterations was used and for the Axilrod-Teller benchmarks the scenario `AutoPas/examples/md-flexible/input/3BodyTest.yaml` running for 500 iterations.

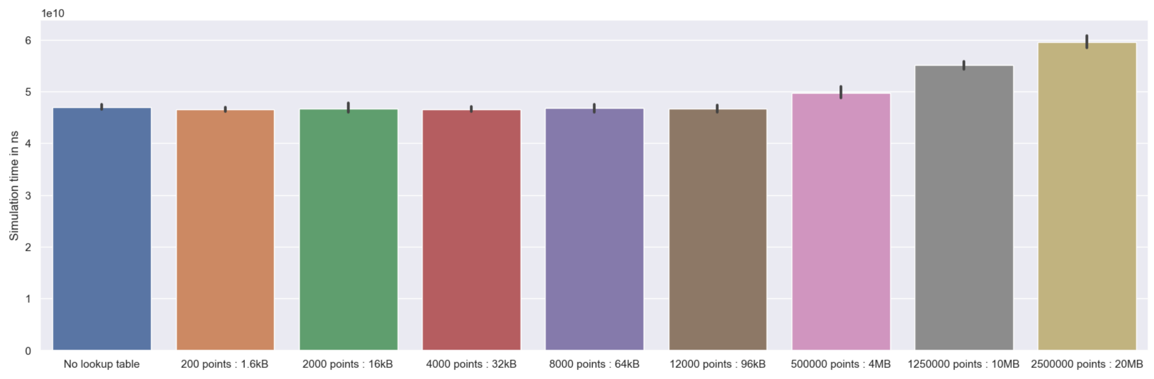


Figure 6.1.: Comparison of simulation time for different Lennard-Jones next neighbor table sizes

Immediately obvious and against expectations in Figure 6.1 is the observation that table size has a lot smaller effect on the performance that what raw latency numbers may lead one to believe. The measurements were made with tables exceeding not only L1 or L2, but even L3 cache and only when the size reached 4MB, overflowing into L3 cache could performance degradation be measured. But even with parts of the table in main memory was the performance far from desastrous. This might be because the most used table entries are kept in faster cache and there are very few misses, although more investigation is required. It is also notable that even though the Lennard-Jones potential is rather cheap to compute the table implementation matched it in performance.

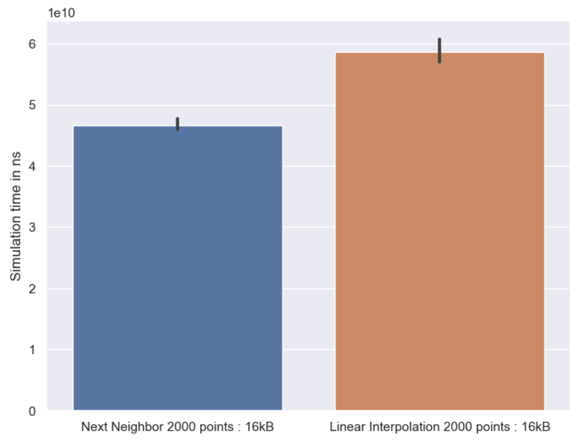


Figure 6.2.: Comparison of simulation time between next neighbor and linear interpolation

Introducing linear interpolation carries a notable performance penalty as seen in Figure 6.2. Obviously the second table lookup introduces additional latency, although it is located on a neighboring cache line, meaning that it most likely is not as heavy as a random lookup. Secondly the calculation itself seems to cause a measurable slow down.

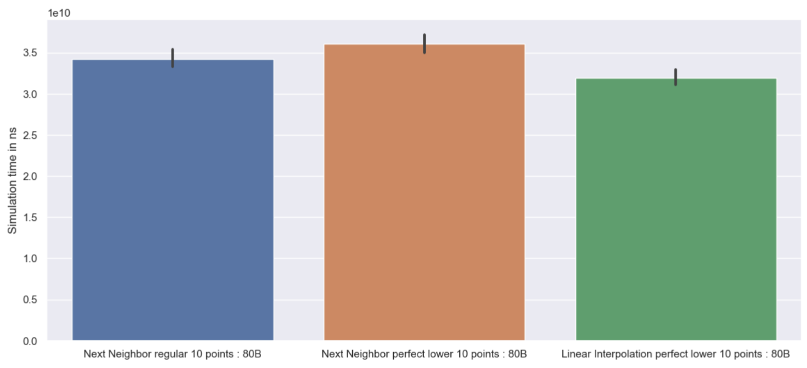


Figure 6.3.: Performance when introducing the ”perfect lower” optimization

For this benchmark the table size was greatly decreased to ensure the lowest interval, which is what has been changed, is hit more often. This had the effect of greatly increasing the performance overall as seen in Figure 6.3, for all implementations, even beyond the regular Lennard-Jones functor. This happens most likely because of compiler optimizations in the table access and extremely good cache locality. This could also explain why introducing this change to the next neighbor version has decreased its speed as the calculation can profit less from these benefits. Extremely interesting is however, that the linear interpolation implementation not only caught up to the other two, but significantly exceeded their performance. While this could very well be due to the sensitivity of the performance of the test system to heat, enough test runs were performed to mostly exclude this possibility. Therefore the low number of intervals must have allowed the compiler to make great improvements to the interpolation function, while the second lookup profited greatly from the improved cache locality.

6.2. Axilrod-Teller

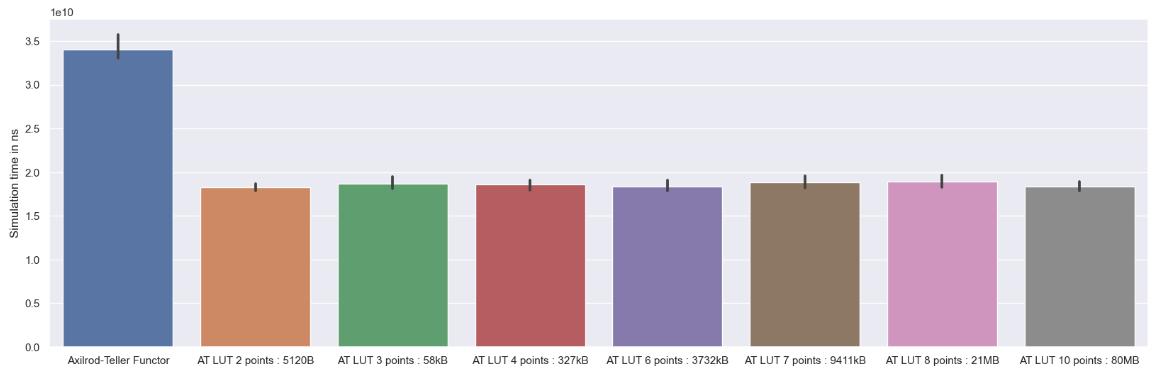


Figure 6.4.: Comparison of simulation time for different Axilrod-Teller next neighbor table sizes

The Axilrod-Teller functor is much more expensive to compute, making the lookup table about twice as fast in Figure 6.4. Again we find that the size of the lookup table is much less important for its performance than what memory latencies would lead one to believe. But due to the extremely high rate of size growth the implementation as it stands is still not a viable replacement for the regular functor. It is however obvious that there is a lot of headroom for interpolation, as done by [Str23], which in combination with a system equipped with a very large amount of memory could make this an option for some users.

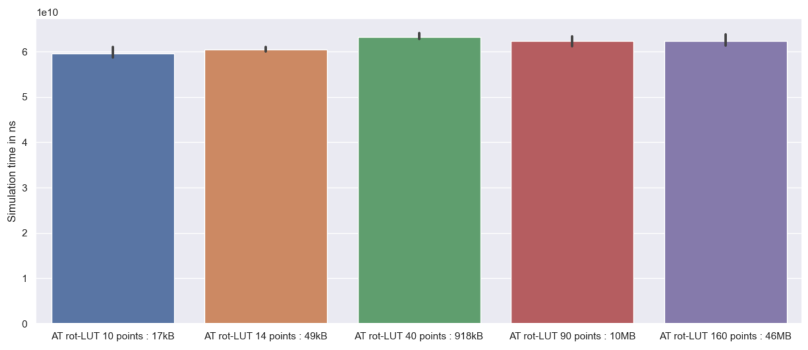


Figure 6.5.: Comparison of simulation time for different Axilrod-Teller triangle table sizes

Firstly, the table size again has little impact on the performance, repeating previous observations. This doesn't deter from the fact that the cost of the rotation operations far exceeds any speedup from utilizing the lookup table, preventing this from being a viable replacement for the Axilrod-Teller functor currently implemented in AutoPas. It may however be an option for even more expensive functors.

7. Future Work

Because the corresponding conditionals would test information already known at the beginning of runtime which doesn't change for the entirety of the execution, they would always evaluate to the same result. If this is recognized by the processor, the performance impact of implementing the lookup tables as regular classes instead of templates.
Additionally the interaction of different types of particles should be implemented. Different interpolation functions are also a possible avenue for exploration and finally an integration into AutoPas' auto-tuning would finalize the process after all other work is completed.
Before this can happen, much more investigation into the performance charasteristics of the different implementations has to be done.

Part III.

Conclusions

The implementation of different lookup table designs has yielded many interesting insights. While for the Lennard-Jones functor the tables could be considered a viable alternative, if the accuracy can be improved enough for the desired application, the Axilrod-Teller functor is too complex to be replaced by any of this work's implementations. Interesting performance characteristics were discovered, but much more investigation is yet to be done and improvements to be made.

Part IV.

Appendix

As part of this work the time to start and stop a timer in AutoPas was determined to average around 4273499891 ns, with around 2019457897 ns being measured by the timer itself due to the time it takes to stop.

List of Figures

List of Tables

Bibliography

[AT43] B. M. Axilrod and E. Teller. Interaction of the van der Waals Type Between Three Atoms. *The Journal of Chemical Physics*, 11(6):299–300, 06 1943.

[Att92] Phil Attard. Simulation results for a fluid with the axilrod-teller triple dipole potential. *Phys. Rev. A*, 45:5649–5653, Apr 1992.

[AW57] B. J. Alder and T. E. Wainwright. Phase Transition for a Hard Sphere System. *The Journal of Chemical Physics*, 27(5):1208–1209, 11 1957.

[BC21] Danilo de Camargo Branco and Gary J. Cheng. Employing hybrid lennard-jones and axilrod-teller potentials to parametrize force fields for the simulation of materials' properties. *Materials*, 14(21), 2021.

[Bor21] Małgorzta Borówko. Special issue on advances in molecular simulation. *International Journal of Molecular Sciences*, 22(22), 2021.

[BZBP13] Hans-Joachim Bungartz, Stefan Zimmer, Martin Buchholz, and Dirk Pflüger. *Modellbildung und Simulation.* Springer Spektrum Berlin, Heidelberg, 10 2013.

[CKCFR03] Martin Campbell-Kelly, Mary Croarken, Raymond Flood, and Eleanor Robson. *The History of Mathematical Tables: From Sumer to Spreadsheets.* Oxford University Press, 10 2003.

[DLR$^+$15] Nathalie Deltimple, Bertrand Le Gal, Chiheb Rebai, Alexis Aulery, Nicolas Delaunay, Dominique Dallet, Didier Belot, and Eric Kerhervé. Chapter 2 - cartesian feedback with digital enhancement for cmos rf transmitter. In Eric Kerhervé and Didier Belot, editors, *Linearization and Efficiency Enhancement Techniques for Silicon Power Amplifiers*, pages 35–53. Academic Press, Oxford, 2015.

[EP10] Peter Eastman and Vijay S. Pande. Efficient nonbonded interactions for molecular dynamics on a graphics processing unit. *Journal of Computational Chemistry*, 31(6):1268–1272, 2010.

[Fog24] A. Fog. *Optimizing software in C++.* 03 2024.

[FS23] D. Frenkel and B. Smit. *Understanding molecular simulation.* Elsevier Science & Technology, 2023.

[FW23] Johann Fischer and Martin Wendland. On the history of key empirical intermolecular potentials. *Fluid Phase Equilibria*, 573:113876, 2023.

[GSBN22] Fabio Alexander Gratl, Steffen Seckler, Hans-Joachim Bungartz, and Philipp Neumann. N ways to simulate short-range particle systems: Automated algorithm selection with the node-level library autopas. *Computer Physics Communications*, 273:108262, 2022.

[GST$^+$19] Fabio Alexander Gratl, Steffen Seckler, Nikola Tchipev, Hans-Joachim Bungartz, and Philipp Neumann. Autopas: Auto-tuning for particle simulations. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 748–757, 2019.

[HLR$^+$23] Binxiao Huang, Jason Chun Lok Li, Jie Ran, Boyu Li, Jiajun Zhou, Dahai Yu, and Ngai Wong. Hundred-kilobyte lookup tables for efficient single-image super-resolution, 2023.

[IBM11] IBM. The ibm 700 series, computing comes to business, 2011. Accessed 2019 - 01 - 05.

[LSH24] Johannes Lenhard, Simon Stephan, and Hans Hasse. A child of prediction. on the history, ontology, and computation of the lennard-jonesium. *Studies in History and Philosophy of Science*, 103:105–113, 2024.

[Mav15] Vasilios Mavroudis. Computing small discrete logarithms using optimized lookup tables. 2015.

[Par19] Behrooz Parhami. Tabular computation. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019.

[PBJB84] E. E. Polymeropoulos, J. Brickmann, L. Jansen, and R. Block. Analysis of three-body potentials in systems of rare-gas atoms: Axilrod-teller versus three-atom exchange interactions. *Phys. Rev. A*, 30:1593–1599, Oct 1984.

[PF05] M. Pharr and R. Fernando, editors. *GPU gems 2*. Addison-Wesley Professional, 2005.

[Qui] J. Quintanilla. Square roots and logarithms without a calculator (part 3). https:meangreenmath.com20130803square-roots-without-a-calculator-part-3. Accessed 12.04.2024.

[RWK$^+$09] Morris Riedel, Felix Wolf, Dieter Kranzlmüller, Achim Streit, and Thomas Lippert. Research advances by using interoperable e-science infrastructures. *Cluster Computing*, 12:357–372, 12 2009.

[Sad98] Richard J. Sadus. Exact calculation of the effect of three-body axilrod–teller interactions on vapour–liquid phase coexistence. *Fluid Phase Equilibria*, 144(1):351–359, 1998.

[SB94] Harshvardhan Sharangpani and Mickey L. Barton. Statistical analysis of floating point flaw in the pentium processor. 1994.

[Ste13] Martin Oliver Steinhauser. *Computer Simulation in Physics and Engineering*. De Gruyter, Berlin, Boston, 2013.

[Str23] P. Ströker. *Bestimmung thermodynamischer Eigenschaften von Fluiden mit einer weiterentwickelten molekularen Simulationsmethodik und hochgenauen ab initio-Potentialen.* PhD thesis, 2023.

[Wil12] Chris Wilcox. *A methodology for automated lookup table optimization of scientific applications.* PhD thesis, USA, 2012. AAI3509633.