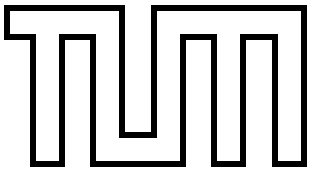


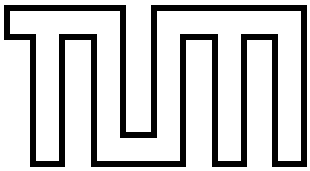
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Fuzzy Tuning Technique for Molecular Dynamics Simulations in AutoPas

Manuel Lerchner



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Exploring Fuzzy Tuning Technique for Molecular Dynamics Simulations in AutoPas

Untersuchung von Fuzzy Tuning Verfahren für Molekulardynamik-Simulationen in AutoPas

Author:	Manuel Lerchner
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisors:	Manish Kumar Mishra, M.Sc. (hons) &
	Samuel James Newcome, M.Sc.
Date:	10.08.2024

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

Munich, 10.08.2024 Manuel Lerchner

Acknowledgements

I extend my sincere gratitude to my advisors, Sam and Manish, for their invaluable guidance and feedback throughout this thesis. Their expertise and insights have been instrumental in shaping the direction of this work, and I greatly appreciated our engaging discussions.

I'm also grateful to the Chair of Scientific Computing and Professor Dr. Hans-Joachim Bungartz for providing me with the opportunity to work on this project. Moreover, I would like to thank the Leibniz Supercomputing Centre for providing the necessary computational resources for this research.

Finally, I wish to express my heartfelt thanks to my family and friends. The unwavering support of my family enabled me to fully focus on my studies, while my friends offered both distractions when I needed breaks and inspiration when I needed motivation. Without them, this journey would have been far less enjoyable.

Abstract

AutoPas is a high-performance, auto-tuned particle simulation library for many-body systems, capable of dynamically switching between algorithms and data structures to guarantee optimal performance throughout the simulation. This thesis introduces a novel fuzzy logic-based tuning strategy for AutoPas, allowing users to guide the tuning process by specifying custom Fuzzy Systems, which can be used to efficiently prune the search space of possible parameter configurations. Efficient tuning strategies are crucial, as they allow for discarding poor parameter configurations without evaluating them, thus reducing tuning time and improving overall library performance.

We demonstrate that a data-driven approach can automatically generate Fuzzy Systems that significantly outperform existing tuning strategies on specific benchmarks, resulting in speedups of up to 1.96x compared to the FullSearch Strategy on scenarios included in the training data and up to 1.35x on scenarios not directly included.

The Fuzzy Tuning Strategy can drastically reduce the number of evaluated configurations during tuning phases while achieving comparable tuning results, making it a promising alternative to the existing tuning strategies.

Zusammenfassung

AutoPas ist eine hochperformante, selbstoptimierende Teilchensimulationsbibliothek für Mehrkörpersysteme, welche in der Lage ist, dynamisch zwischen verschiedenen Algorithmen und Datenstrukturen zu wechseln, um eine optimale Leistung während der Simulation zu gewährleisten. In dieser Arbeit wird eine neuartige, auf Fuzzy-Logik basierende Tuning-Strategie für AutoPas vorgestellt, die es dem Benutzer ermöglicht, Tuning-Phasen durch die Vorgabe von benutzerdefinierten Fuzzy-Systemen zu steuern, um so den Suchraum möglicher Parameterkonfigurationen effizient zu verkleinern. Solche effizienten Suchstrategien sind von entscheidender Bedeutung für AutoPas, da sie es ermöglichen, schlechte Parameterkonfigurationen auszuschließen, ohne sie zu evaluieren, wodurch die Tuning-Zeit reduziert und die Gesamtleistung der Bibliothek verbessert wird.

Wir zeigen, dass ein datengesteuerter Ansatz zur automatischen Generierung von Fuzzy-Systemen in bestimmten Tests eine deutlich bessere Leistung als bestehende Tuning-Strategien erbringen kann. Im Vergleich zur FullSearch-Strategie kann die Fuzzy-Tuning-Strategie eine Geschwindigkeitssteigerung von bis zu 1,96x bei Szenarien aus den Trainingsdaten und bis zu 1,35x bei Szenarien, die nicht direkt in den Trainingsdaten enthalten sind, erzielen.

Die Fuzzy-Tuning-Strategie kann die Anzahl der evaluierten Konfigurationen während der Tuning-Phasen drastisch reduzieren, während sie dennoch vergleichbare Tuning-Ergebnisse erzielt, was sie zu einer vielversprechenden Alternative zu den bestehenden Tuning-Strategien macht.

Contents

1. Introduction

Molecular Dynamics (MD) is a computational method used to simulate the behavior of atoms and molecules over time. In recent years, MD simulations have become essential in many scientific fields, including chemistry, physics, biology, and materials science. Such simulations are used to study various systems, ranging from simple gases and liquids to complex biological molecules and new materials.

MD simulations act on an atomic level and attempt to explain macroscopic properties of a system from the interactions between the individual atoms and molecules. The recent advances in computational power have made it possible to simulate systems with millions of particles over long time scales, allowing researchers to study complex systems in unprecedented detail. Contrary to experimental methods, MD simulations can provide detailed information about the behavior of atoms and molecules, sometimes inaccessible to experimental methods [PS17].

Two illustrations of such simulations are shown in Figure 1.1 and Figure 1.2. The first image shows a simulation of the HIV-1 capsid, a protein shell that surrounds the genetic material of the human immunodeficiency virus (HIV). Using a simulation-based approach, researchers could study critical properties of the HIV-1 capsid, which would be difficult to access using other methods [PS17]. The second image shows a simulation of shear band formation around a precipitate in metallic glass. This simulation found evidence that depending on the precipitate size, shear bands can either dissolve, wrap around, or be blocked by the precipitate [BPR+16]. This information is crucial for understanding the mechanical properties of metallic glasses and can be used to design new materials with improved properties.

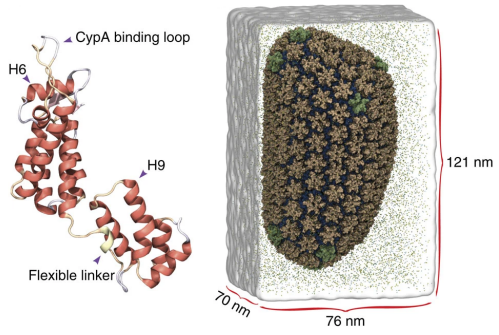


Figure 1.1.: MD simulation with 64,423,983 atoms of the HIV-1 capsid. Perilla et al. [PS17] investigated properties of the HIV-1 capsid at an atomic resolution.

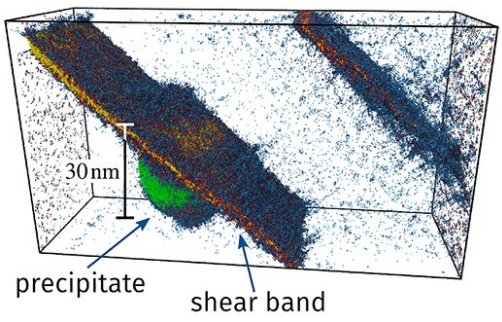


Figure 1.2.: MD simulations of shear band formation around a precipitate in metallic glass, as demonstrated by Brink et al. [BPR+16].

Simulating such systems is very computationally demanding and typically requires the use of high-performance computing (HPC) systems for large-scale simulations. Another challenge is the development of efficient simulation software that can handle the complexity of the systems and make optimal use of the available computational resources. This thesis focuses on the development of AutoPas, a high-performance, auto-tuned particle simulation library for many-body systems, which tries to address these challenges by dynamically switching between algorithms and data structures to guarantee high performance throughout the simulation.

This switching mechanism in AutoPas is guided by so-called *tuning strategies*, which explore the space of available algorithms and data structures and attempt to find suitable configurations for the current simulation state. The development of efficient tuning strategies is crucial, as efficient choices can significantly reduce the total runtime of the simulation, making MD simulations more accessible to researchers and enabling the study of more complex systems.

In particular, we develop a novel fuzzy logic-based tuning strategy, which allows users to encode their domain knowledge in the tuning process. Furthermore, we investigate a data-driven approach to automatically generate fuzzy systems, and we will show that the proposed fuzzy tuning strategy can outperform existing tuning strategies on specific benchmarks.

2. Theoretical Background

2.1. Molecular Dynamics

2.1.1. Quantum Mechanical Background

Our current knowledge of physics suggests that the behavior of atoms and molecules is governed by the laws of quantum mechanics, where particles are described by probabilistic wave functions evolving over time. In 1926, Austrian physicist Erwin Schrödinger formulated a mathematical model describing this concept, which has since gained widespread acceptance and is now generally known as the Schrödinger equation. The Schrödinger equation is a partial differential equation describing the time evolution of a quantum system and is given by:

$$i\hbar\frac{\partial\Psi(\vec{r},t)}{\partial t} = \hat{H}\Psi(\vec{r},t) \tag{2.1}$$

Here $\Psi(\vec{r},t)$ is the system's wave function, evolving over time t and space $\vec{r}$. $\hat{H}$ is the Hamiltonian operator describing the system's energy, i is the imaginary unit, and $\hbar$ is the reduced Planck constant.

The Schrödinger equation provides a way to calculate the future states of a quantum system given the system's current state. However, the computational complexity of solving this equation increases dramatically with the number of particles involved and quickly becomes infeasible for systems with more than a few particles [LM15]. To illustrate this complexity, consider simulating a single water molecule. This molecule consists of three nuclei (two hydrogen atoms and one oxygen atom) and 10 electrons. Each of these 13 objects requires three spatial coordinates to describe its position, resulting in a total of $(2+1+10)\times 3 = 39$ variables. Following [LM15], the Schrödinger equation for this system can be written as:

$$i\hbar\frac{\partial\Psi}{\partial t} = -\hbar^2\sum_{i=1}^{13}\frac{1}{2m_i}\left(\frac{\partial^2\Psi}{\partial x_i^2}+\frac{\partial^2\Psi}{\partial y_i^2}+\frac{\partial^2\Psi}{\partial z_i^2}\right)+U_p(x_1,y_1,z_1,\ldots,x_{13},y_{13},z_{13})\Psi \tag{2.2}$$

In this equation, m_i is the mass of the i-th object, x_i, y_i, and z_i are the spatial coordinates of the i-th object, and U_p is the potential energy function of the system.

As the Schrödinger equation is a partial differential equation, it is computationally expensive to solve for systems with many particles, as one quickly runs into the curse of dimensionality. Larger systems, such as the HIV-1 capsid shown in Figure 1.1 consisting of millions of atoms, are practically impossible to simulate using the Schrödinger equation directly.

Luckily, the Born-Oppenheimer approximation simplifies the Schrödinger equation so that it becomes computationally feasible to simulate even large systems of particles. The approximation exploits the significant mass difference between electrons and nuclei[1], making it possible to solve both motions independently [ZBB+13]. As the forces acting on the heavy nuclei cause way slower movements compared to the same force acting on the electrons, it is possible to approximate the position of the nuclei as entirely stationary. This simplification yields a new potential energy function U combining all electronic and nuclear energies, which only depends on the nuclei's positions. Using this simplification, efficient simulations of systems with many particles become possible.

As the Born-Oppenheimer approximation is based on simplifications of the full model, it is not always accurate. Depending on the system under investigation and the chosen potential energy function U, the Born-Oppenheimer approximation may neglect specific quantum mechanical effects, resulting in inaccuracies in the simulation.

Despite these limitations, the Born-Oppenheimer approximation is widely used in molecular dynamics simulations and is the best-known method to simulate systems with many particles.

2.1.2. Classical Molecular Dynamics

After applying the Born-Oppenheimer approximation and using Newton's second law of motion, the Schrödinger equation can be transformed into a system of ordinary differential equations of the form:

$$m_i \frac{d^2 \vec{r}_i}{dt^2} = -\nabla_i U \tag{2.3}$$

Where m_i is the mass of the i-th particle, $\vec{r}_i$ is the position of the i-th particle, and U is the potential energy function of the system. These equations precisely describe a classical particle system, where particles are treated as point masses moving through space under the influence of forces. The forces are derived from the potential energy function U and are calculated using the negative gradient of the potential energy function $\nabla_i U$.

2.1.3. Potential Energy Function

The potential energy function U is a critical component of molecular dynamics simulations as it fundamentally defines the properties of the system. MD simulations use many different potential energy functions, all of which are tailored to describe specific aspects of the system. Those potentials typically use a mixture of 2-body, 3-body, and 4-body interactions between the particles, each used to describe different aspects of particle interactions. The 2-body interactions typically express the effect of Pauli repulsion, atomic bonds, and coulomb interactions, while higher-order interactions allow for asymmetric wave functions for atoms in bound-groups [LM15].

A common choice for the potential energy function is the Lennard-Jones potential. This potential can reproduce the potential energy surfaces of many biological systems [Phy] while still being very simple and efficient to compute. It is a pairwise potential that describes the

[1] The mass ratio of a single proton to an electron is approximately 1836:1, illustrating the vast difference in mass between nuclei and electrons.

interaction between two particles and mainly emulates the attractive Van-der-Waals forces and the repulsive Pauli repulsion forces between the particles [Che].

The Lennard-Jones potential is given by:

$$U_{LJ}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.4}$$

Where r is the distance between the particles, ϵ is the depth of the potential well, and σ is the distance at which the potential is zero. The parameters ϵ and σ can differ for each type of particle interaction and are either determined from theoretical considerations of the material or chosen to match experimental data [MAMM20] [Iri21].

2.1.4. Numerical Integration

Since the simulation domain potentially consists of a vast number of particles all interacting with each other, it is generally not possible to solve the equations of motion analytically. This problem is known under the N-body problem, and it can be shown that there are no general solutions for systems with more than two particles. It is, however, possible to approximate solutions of these equations of motion using numerical integration methods. A widely used method for this purpose is the Velocity-Störmer-Verlet algorithm. It is defined as follows:

$$\vec{r}_i(t + \Delta t) = \vec{r}_i(t) + \Delta t \cdot \vec{v}_i(t) + (\Delta t)^2 \frac{\vec{F}_i(t)}{2m_i} \tag{2.5}$$

$$\vec{F}_i(t + \Delta t) = -\nabla U(\vec{r}_i(t + \Delta t)) \tag{2.6}$$

$$\vec{v}_i(t + \Delta t) = \vec{v}_i(t) + \Delta t \frac{\vec{F}_i(t) + \vec{F}_i(t + \Delta t)}{2m_i} \tag{2.7}$$

Here $\vec{r}_i(t)$ is the position, $\vec{v}_i(t)$ is the velocity, $\vec{F}_i(t)$ is the force acting on the i-th particle at time t. Δt is the time step size, and m_i is the mass of the i-th particle.

2.1.5. Simulation Loop

By using the ideas introduced above, it is possible to simulate the behavior of systems of particles over time. The general simulation loop for a molecular dynamics simulation can be divided into the following steps:

Algorithm 1: Molecular Dynamics Simulation Loop (Velocity-Störmer-Verlet)

1 **Initialize** particle positions and velocities based on initial conditions
2 **while** *simulation time* $<$ *desired time* **do**
3 Update all particle positions using Equation 2.5
4 Calculate all forces $\vec{F}_i(t + \Delta t)$ at the new positions using Equation 2.6
5 Update all particle velocities using Equation 2.7
6 Apply external forces or constraints (if any)
7 Increment simulation time
8 **end**

Many different software packages exist to perform such simulations. Some widely used examples are LAAMPS[2] and GROMACS[3]. Both attempt to efficiently solve the underlying N-body problem and provide the user with a high-level interface to specify the parameters and properties of the simulation.

Different algorithms and implementation possibilities exist to simulate such systems efficiently. Typically, no single best approach works well for all scenarios and environments, as the optimal implementation heavily depends on the simulation state and the hardware used to perform the simulation. LAAMPS and GROMACS use a single (though highly optimized) implementation, which may result in suboptimal performance for some simulation states.

In the following section, we will introduce AutoPas, a library designed to address this issue by automatically switching between different implementations to guarantee optimal performance.

2.2. AutoPas

AutoPas is an open-source library designed to achieve optimal node-level performance for short-range particle simulations. On a high level, AutoPas can be seen as a black box performing arbitrary N-body simulations with short-range particle interactions. However, AutoPas differentiates itself from other libraries by providing many algorithmic implementations for the N-body problem, each with different performance and memory usage trade-offs. AutoPas can automatically switch between these implementations to guarantee optimal performance throughout the simulation.

Since AutoPas only provides the infrastructure to perform arbitrary N-body simulations, the user is tasked with implementing the actual simulation logic on top of AutoPas. Fortunately, AutoPas is equipped with some example applications, such as `md_flexible`. `md_flexible` is a molecular dynamics framework built on top of AutoPas that allows users to specify and run arbitrary molecular dynamics simulations.

In this thesis, we will primarily focus on the tuning of `md_flexible` simulations, but the concepts can be easily transferred to other applications built on top of AutoPas.

2.2.1. Autotuning in AutoPas

AutoPas internally alternates between two phases of operation. The first phase is the *tuning phase*, where AutoPas tries to find the best configuration of parameters that minimize a chosen performance metric, such as time or energy usage. This is achieved by trying out different configurations of parameters and measuring their performance. The configuration that optimizes the chosen performance metric is then used in the following *simulation phase*, assuming that the optimal configuration found in the tuning phase still works well in the following simulation steps.

As the simulation progresses and the characteristics of the system change, the previously chosen configuration can drift away from the actual optimal configuration [GSBN21]. To

[2] `https://lammps.sandia.gov/`
[3] `https://www.gromacs.org/`

counteract this, AutoPas periodically alternates between tuning and simulation phases to ensure that the used configuration remains close to optimal during the entire simulation.

The power of AutoPas comes from its vast amount of tunable parameters and the enormous search space associated with them. In the following sections, we will discuss all currently tunable parameters in AutoPas and briefly present the tuning strategies that are available to efficiently explore the search space.

2.2.2. Tunable Parameters

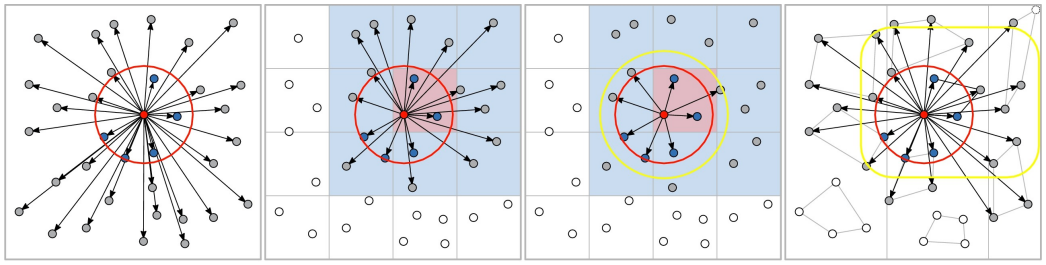
AutoPas currently provides six tunable parameters, which can mostly[4] be combined freely. A collection of parameters is called a *Configuration*. Each configuration consists of the following parameters:

1. **Container Options:**
   The container options are related to the data structure used to store the particles. The most important categories of data structures in this section are:
   a) **DirectSum**
   DirectSum does not use any additional data structures to store the particles. Instead, it simply holds a list of all particles. Consequently, it needs to rely on brute-force calculations of the forces between all pairs of particles and requires $O(N^2)$ distance checks in each iteration. This inferior complexity renders it completely useless for larger simulations.
   Generally should not be used except for tiny systems or demonstration purposes. [VBC08]
   b) **LinkedCells**
   LinkedCells segments the domain into a regular cell grid and only considers interactions between particles from neighboring cells. This results in the trade-off that particles further away are not considered for the force calculation. In practice, this is not a big issue, as all short-range forces drop off quickly with distance anyway. LinkedCells also provides a high cache hit rate as particles inside the same cell can be stored contiguously in memory. Typically, the cell size is chosen to equal the force cutoff radius r_c, meaning each particle only needs to check interactions between particles inside the $3 \times 3 \times 3$ cell grid around the current cell. All other particles are guaranteed to be further away than the cutoff radius r_c and cannot contribute to the acting forces. This reduction in possible interactions typically results in a complexity of just $O(N)$ distance checks in each iteration, assuming the particles are spread evenly. However, there is room for improvement as the constant overhead factor can be pretty high, as most distance checks performed by LinkedCells still do not contribute to the force calculation when using $r_{cell} = r_c$. This is caused by many particles inside the $3 \times 3 \times 3$ cell grid being still further away than the cutoff radius [GST+19].
   However, still generally good for large, homogeneous[5] systems.

[4] There are some exceptions as some choices of parameters are incompatible with each other.
[5] Homogeneous in this context, the particles are distributed evenly across the domain.

c) **VerletLists**
VerletLists are another approach to creating neighbor lists for the particles. Contrary to LinkedCells, VerletLists do not rely on a regular grid but instead use a spherical region around each particle to determine its relevant neighbors. The algorithm creates and maintains a list of all particles present in a sphere within radius $r_c \cdot s$ around each particle, where r_c is the cutoff radius and $s > 1$ is the skin factor allowing for a buffer zone around the cutoff radius. By choosing a suitable buffer zone, such that no fast-moving particle can enter the cutoff radius unnoticed, it is possible to only recalculate the neighbor list every few iterations. This method can also result in a complexity of $O(N)$ distance checks in each iteration and can, depending on the skin size s, achieve a lower constant overhead factor compared to LinkedCells, as a higher percentage of distance checks contribute to the force calculation [GST+19]. Ideally, the buffer size should be as small as possible, resulting in very few unnecessary distance checks. This, however, means that the neighbor lists need to be updated more frequently, or the simulation needs to be run at higher temporal precision to prevent particles from entering or leaving the cutoff radius unnoticed. Finding a good skin factor s is crucial for the performance of VerletLists.
Generally good for large systems with high particle density.

d) **VerletClusterLists**
VerletClusterLists differ from regular VerletLists in the way the neighbor lists are stored. Instead of storing the neighbor list for each particle separately, $n_{cluster}$ particles are grouped into a so-called *cluster*, and a single neighbor list is created for each cluster. This reduces memory overhead as the neighbor list only needs to be stored once for each cluster. Whenever two clusters are close, all interactions between the particles in the two clusters are calculated. This also results in a complexity of $O(N)$ distance checks in each iteration but provides the advantage of greatly reduced memory usage compared to regular VerletLists.
Generally suitable for large systems with high particle density



(a) Direct Sum (b) Linked Cells (c) Verlet Lists (d) Verlet Cluster Lists

Figure 2.1.: Visualization of different container options. Source: Gratl et al. [GSBN21]

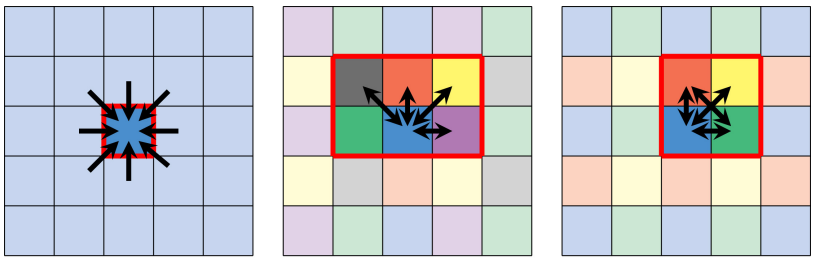
2. **Load Estimator Options:**
The Load Estimator Options relate to how the simulation behaves in a parallelized setting. As this thesis primarily focuses on tuning single-node performance, we will not go into detail about these options.

3. **Traversal Options:**
   The Traversal options specify the order in which the particles are visited when calculating the forces between them. Especially in parallelized settings, the traversal options can significantly impact the simulation's performance. Different settings can allow more workers to work simultaneously, with varying synchronization overhead. Some avaialble traversal options are:

   a) **Sliced Traversals**
   Sliced Traversals divide the domain into different slices, each processed by a different thread. The iteration order inside the slices is ideally chosen so that two threads are unlikely to work simultaneously on cells sharing common neighbors. As data races can occur at boundaries, the slices must be synchronized using a single lock, resulting in only a minor synchronization overhead [GSBN21].

   b) **Colored Traversal**
   Colored traversals assign a color to each cell in the container so that no same-colored cells share a common neighbor. During the traversal, all threads work on cells of the same color simultaneously, allowing for a high degree of parallelism without the need for synchronization. Some available colorings are:

   - **C01**
     The C01 traversal only uses a single color. This method is embarrassingly parallel but comes at the cost of being incompatible with the Newton 3 optimization, as there is no way of preventing data races between neighboring cells. However, it provides perfect parallelism and no overhead.
   - **C18**
     The C18 traversal is a more sophisticated way of coloring the domain. The domain is divided into 18 colors, as depicted in Figure 2.2. It is compatible with the Newton 3 optimization, as the coloring prevents simultaneous access to neighboring cells.
   - **C08**
     The C08 traversal is similar to the C18 traversal but only uses eight colors. Due to the fewer colors, fewer passes are needed to complete the traversal, resulting in a lower synchronization overhead. A high degree of parallelism can still be achieved and is compatible with the Newton 3 optimization.



(a) C01 (b) C18 (c) C08

Figure 2.2.: Visualization of color-based traversal options. Source: Newcome et al. [NGNB23]

4. **Data Layout Options:**
   The Data Layout Options determine how the particles are stored in memory. The two possible data layouts are:
   a) **SoA**
      The SoA (Structure of Arrays) data layout stores the particles' properties in separate arrays. For example, all particles' x-, y- and z-positions are stored in separate arrays. This data layout is beneficial for vectorization as the properties of the particles are stored contiguously in memory and can be loaded into vector registers using a single instruction. However, the SoA data layout can lead to inefficient cache utilization as the properties of the same particle are not stored close to each other in memory.
   b) **AoS**
      The AoS (Array of Structures) data layout stores all particle properties in different structures. This allows for efficient cache utilization when working on particles, as all properties are close to each other in memory. Loading each particle's properties into vector registers is more complicated, as the properties are not stored contiguously in memory. This can lead to inefficient vectorization of the force calculations.
5. **Newton 3 Options:**
   The Newton 3 option controls an optimization technique to reduce the number of force calculations. Newton's third law states that for every action, there is an equal and opposite reaction, which means that the magnitude of the force between two particles is the same, regardless of which particle is the source and which is the target. This rule can be exploited to reduce the number of force calculations by a factor of 2, as one particle can apply the negation of its experienced force to the other particle.
   a) **Newton3 Off**
      If Newton 3 is turned off, the forces between all pairs of particles are calculated twice, once for each particle. This results in a constant overhead of factor 2.
   b) **Newton3 On**
      If Newton 3 is turned on, the forces between all pairs of particles are calculated only once. There is no more overhead due to recalculating the forces twice, but turning on Newton 3 requires additional bookkeeping, especially in multi-threaded environments. This results in more complicated traversal algorithms.
      Generally should be turned on whenever available.
6. **Cell Size Factor:**
   The Cell Size Factor is a parameter that is used to determine the size of the cells in the LinkedCells container[6]. If the cell size factor is set to high, many spurious distance checks are performed, as many particles further away than the cutoff radius are considered for the force calculation. Therefore, it is beneficial to reduce the cell size factor. However, the increased overhead of managing more cells can quickly offset the performance gain, and a trade-off between the two needs to be found.

[6]The option is also relevant for other containers such as VerletLists as those configurations internally also build their neighbor lists using a Cell Grid

2.2.3. Tuning Strategies

Tuning strategies are the heart of AutoPas, as they fundamentally guide the tuning phases. Their goal is to find a good combination of tunable parameters for the current simulation state without spending too much time exploring the search space.

AutoPas provides a couple of different tuning strategies that are used to explore the search space of possible configurations. The tuning strategies differ in how they explore the search space and how they decide which configurations to test next. However, all strategies follow the same general pattern: They first select a queue of promising configurations to test. All those configurations are then evaluated for a few iterations in the actual simulation, and relevant statistics are collected. In between the iterations, the tuning strategy can use the intermediate performance data to update the configuration queue if desired. After all worthwhile configurations have been tested, the best-performing configuration is chosen for the following simulation phase.

The currently available tuning strategies in AutoPas are:

1. **Full Search**
   The Full Search strategy is the default tuning strategy in AutoPas. It selects all possible combinations for being tested. As all possible configurations are evaluated, finding the best configuration for the current simulation state is guaranteed. However, it is typically very costly in terms of time and resources as it has to spend a lot of time simulating bad parameter combinations. This is a big issue as the number of possible parameter combinations grows exponentially with the number of parameters, and many of them potentially perform very poorly. This makes the full search approach infeasible, especially if more tunable options are added to AutoPas.
2. **Random Search**
   The Random Search strategy is a simple tuning strategy that randomly samples a given number of configurations from the search space. This approach is faster than the Full Search strategy as it does not need to test all possible combinations of parameters. However, it does not guarantee to find the best parameters for the current simulation state.
3. **Predictive Tuning**
   The Predictive Tuning strategy attempts to extrapolate previous measurements to predict how the configuration would perform in the current simulation state. It filters the search space and only keeps configurations predicted to perform reasonably well. The extrapolations are accomplished using methods such as linear regression or constructing polynomial functions through the previous measurements.
4. **Bayesian Search**
   Two implementations of Bayesian tuning exist in AutoPas. Those methods apply Bayesian optimization techniques to predict suitable configurations using performance evidence from previous measurements.
5. **Rule Based Tuning**
   The Rule Based Tuning strategy uses a set of predefined rules to automatically filter out configurations that are expected to perform poorly. The rules are built on expert knowledge and could look like this:

```
if numParticles < lowNumParticlesThreshold:
      [dataLayout="AoS"] >= [dataLayout="SoA"] with same
            container, newton3, traversal, loadEstimator;
endif
```

The rule states that the data layout "AoS" is generally better than "SoA" if the number of particles is below a certain threshold. The rule-based tuning method can be very effective if the rules are well-designed.

This thesis aims to extend these tuning strategies with a new approach based on Fuzzy Logic. Conceptually, this new fuzzy logic-based tuning strategy is very similar to the rule-based tuning strategy as it uses expert knowledge encoded in fuzzy rules to prune the search space. However, contrary to classical rules, fuzzy logic can deal with imprecise and uncertain information, which allows it to only partially activate rules depending on the *degree of truth* of the condition. All the suggestions can then be combined based on their degree of activation rather than just following the binary true/false logic. This allows for a more nuanced approach and allows the tuning strategy to interpolate the effect of many different rules to choose the best possible configuration, even if there is no direct rule for this specific case.

2.3. Fuzzy Logic

Fuzzy Logic is a mathematical framework that allows for reasoning under uncertainty. It is an extension of classical logic and extends the concept of binary truth values ($true$ and $false$) to a continuous range of truth values in the interval $[0, 1]$. Instead of just having true or false statements, it is now possible for statements to be, for example, 60% true. This concept is beneficial when modeling human language, as the words tend to be imprecise. For example, *hot* can mean different things to different people. For some people, a temperature above 30° Celsius might be considered *hot*, while for others, only a temperature above 40° Celsius might be considered *hot*. There is no clear boundary between what is considered hot and what is not, but rather a gradual transition between the two. Fuzzy Logic allows modeling such gradual transitions by assigning a degree of truth to each statement.

2.3.1. Fuzzy Sets

Mathematically, the concept of Fuzzy Logic is based on Fuzzy Sets. A Fuzzy Set is a generalization of a classical set where an element can lie somewhere between being a set member and not being a member. Instead of having a binary membership function that assigns a value of 1 to elements that are members of the set and 0 to elements that are not, elements in a fuzzy set have an arbitrary degree of membership in the set. This continuous membership function extends the binary concepts of $\{0, 1\}$ to the continuous interval $[0, 1]$.

Formally a fuzzy set $\tilde{A}$ over a crisp/classical set X is defined by a membership function

$$\mu_{\tilde{A}} : X \to [0, 1] \tag{2.8}$$

which assigns each element $x \in X$ a degree of membership in the interval $[0, 1]$. The classical element-of operator could be written in this style as $\in_A : X \to \{false, true\}$.

The shape of the membership function can be chosen freely and depends on the specific application. However, typical choices involve triangular, gaussian, or sigmoid-shaped functions, depending on whether the value represents an open or closed region. An example of fuzzy sets for the age of a person is shown in Figure 2.3.

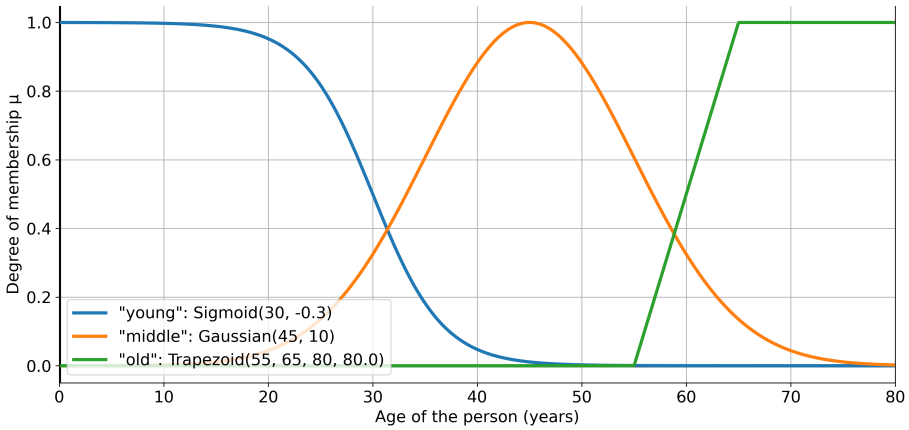


Figure 2.3.: Example of fuzzy sets for the age of a person. Fuzzy sets can be used to model the gradual transition between age groups. The *distributions* could be derived from survey data on how people perceive age groups. In this example, most people would consider a person middle-aged if they are between 35 and 55.

2.3.2. Fuzzy Logic Operations

As fuzzy Sets are a generalization of classical sets, they also need to support adapted versions of the classical set operations of union, intersection, and complement. Those operations need to maintain the semantics of the operation on classical sets and extend them to the continuous interval $[0, 1]$.

The extension of classical operators to fuzzy sets uses so-called De Morgan Triplets. Such a triplet $(\top, \bot, \neg)$ consists of a t-norm $\top : [0,1] \times [0,1] \to [0,1]$, a t-conorm $\bot : [0,1] \times [0,1] \to [0,1]$ and a strong complement operator $\neg : [0,1] \to [0,1]$. Those operators generalize the classical logical operators, which are only defined on the binary truth values $\{true, false\}$ to continuous values from the continuous interval $[0, 1]$. $\top$ generalizes the logical AND operator, $\bot$ generalizes the logical OR operator, and $\neg$ generalize the logical NOT operator. Instead of the binary functions used in classical logic, those new operators are continuous functions implementing mappings between degrees of truth.

The binary operators $\top$ and $\bot$ are often written in infix notation as $a \top b$ and $a \bot b$, similar to how classical logical operators are written.

For a t-norm $\top$ to be valid, it needs to satisfy the following properties:

$$
\begin{aligned}
a \top b &= b \top a && && \text{(Commutativity)} \\
a \top b &\leq c \top d && \text{if } a \leq c \text{ and } b \leq d && \text{(Monotonicity)} \\
a \top (b \top c) &= (a \top b) \top c && && \text{(Associativity)} \\
a \top 1 &= a && && \text{(Identity Element)}
\end{aligned}
$$

A strong complement operator $\neg$ needs to satisfy the following properties:

$$\begin{aligned} \neg 0 &= 1 && && \text{(Boundary Conditions)} \\ \neg 1 &= 0 && && \text{(Boundary Conditions)} \\ \neg y &\leq \neg x && \text{if } x \leq y && \text{(Monotonicity)} \\ \neg(\neg x) &= x && && \text{(Involution)} \end{aligned}$$

The default negation operator in fuzzy logic is $\neg x = 1 - x$. This negation operator satisfies all the abovementioned properties and is the most common choice in practice. In the following sections, we will only consider this standard negation operator.

As in classical logic, the t-conorm $\perp$ can be expressed using $\top$ when applying the generalized De Morgan's laws. De Morgan's laws state that $a \vee b = \neg(\neg a \wedge \neg b)$ for classical logic, which results in $\perp(a, b) = 1 - \top(1 - a, 1 - b)$ for fuzzy logic. Consequently, the properties of the t-conorm can be expressed using the t-norm's properties and omitted here for brevity.

Some common choices for t-norms and t-conorms used in practice are shown in Table 2.1.

Name	t-norm $a \top b$	Corresponding t-conorm $a \perp b$
Min/Max	$\min(a, b)$	$\max(a, b)$
Algebraic	$a \cdot b$	$a + b - a \cdot b$
Einstein	$\frac{a \cdot b}{2-(a+b-a \cdot b)}$	$\frac{a+b}{1+a \cdot b}$
Lukasiewicz	$\max(0, a + b - 1)$	$\min(1, a + b)$

Table 2.1.: Common t-Norms and corresponding t-Conorms concerning the standard negation operator $\neg x = 1 - x$

With these choices of t-norms, t-conorms, and negation operators, it is possible to define the classical set operations of union, intersection, and complement for fuzzy sets. We will only consider the minimum t-norm and maximum t-conorm in the following sections as they are the most common choices in practice. However, we included a comparison of different t-norms and their effect on the intersection operation in Figure 2.4.

- **Intersection**
  By expanding the definition of the classical set operation $\cap$ using its boolean form $x \in A \cap B \iff x \in A \wedge x \in B$, we can directly translate this to the fuzzy set intersection operation using the t-norm $\top$. The resulting membership function is given by $\mu_{\tilde{A} \cap \tilde{B}}(x) = \mu_{\tilde{A}}(x) \top \mu_{\tilde{B}}(x)$. Using the minimum t-norm, the intersection of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is described by the following membership function:

$$\mu_{\tilde{A} \cap \tilde{B}}(x) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x))$$

- **Union**
  By expanding the definition of the classical set operation $\cup$ using its boolean form $x \in A \cup B \iff x \in A \vee x \in B$, we can directly translate this to the fuzzy set union operation using the t-conorm $\perp$. The resulting membership function is given

by $\mu_{\tilde{A}\cup\tilde{B}}(x) = \mu_{\tilde{A}}(x) \perp \mu_{\tilde{B}}(x)$. Using the maximum t-conorm, the union of two fuzzy sets $\tilde{A}$ and $\tilde{B}$ is described by the following membership function:

$$\mu_{\tilde{A}\cup\tilde{B}}(x) = \max(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(x))$$

- **Complement**
  By again expanding the definition of the classical set operation A^c using its boolean form $x \in A^c \iff \neg(x \in A)$, we can directly translate this to the fuzzy set complement operation using the fuzzy negation operator $\neg$. The resulting membership function is given by $\mu_{\tilde{A}^c}(x) = \neg\mu_{\tilde{A}}(x)$. Using the standard negation operator, the complement of a fuzzy set $\tilde{A}$ is described by the following membership function:

$$\mu_{\tilde{A}^c}(x) = 1 - \mu_{\tilde{A}}(x)$$

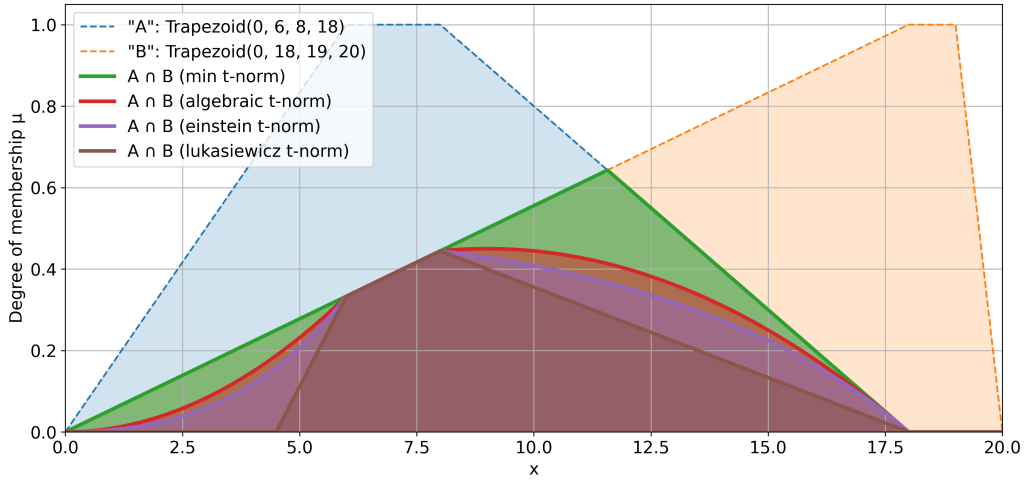


Figure 2.4.: Effect of different t-norms on the intersection of two fuzzy sets $\tilde{A}$ (Blue) and $\tilde{B}$ (Orange). Each membership value is calculated as $\mu_{\tilde{A}\cap_i\tilde{B}}(x) = \mu_{\tilde{A}}(x) \top_i \mu_{\tilde{B}}(x)$ for different t-norms $\top_i$. We can see that the choice of t-norm significantly affects the resulting membership function, with the minimum t-norm resulting in the most generous intersection.

2.3.3. Linguistic Variables

Linguistic variables collect multiple fuzzy sets defined over the same crisp set X into a single object. This variable then allows us to reason about the possible states of the variable more naturally. Contrary to their classical counterparts, linguistic variables do not take a precise numerical value but rather a vaguely defined linguistic term. For example, all fuzzy sets depicted in Figure 2.3 form the linguistic variable *age* with the possible values *young*, *middle-aged*, and *old*. Instead of precisely stating a person's age using a numerical value, we can now declare the age of a person to a mixture of those terms, with each term having a degree of membership in the interval $[0, 1]$. For example, a person of age 35 would be considered 20% young, 60% middle-aged, and 0% old.

2.3.4. Fuzzy Logic Rules

Fuzzy Logic Rules are a way to encode expert knowledge into a Fuzzy Logic system. The rules specify the relationship between the system's input and output variables. The rules are typically encoded in a human-readable way and are written as *IF antecedent THEN consequent* where both the antecedent and the consequent are fuzzy sets. The antecedent is a condition that must be satisfied for the rule to be applied, while the consequent is the action taken if the rule is applied. Since we are not dealing with binary truth values, it is possible that the antecedent is only partially satisfied, which causes the rule to be only partially activated. Consequently, the effect of the consequent is also only partially applied.

The antecedent of the rule can be arbitrarily complicated and may consist of multiple fuzzy sets connected with fuzzy logic operators. The consequent is typically a single fuzzy but could theoretically also be arbitrarily complicated. For this thesis, we will only consider rules with a simple assignment as their effect. Therefore, the general form of a fuzzy rule is:

$$\begin{aligned}
\text{FuzzyRule} &::= \text{IF FuzzySet THEN LinguisticVariable} = \tilde{A} && \text{(Rule)} \\
\text{FuzzySet} &::= (\text{FuzzySet}) && \text{(Parentheses)} \\
&\quad \mid \text{FuzzySet AND FuzzySet} && \text{(Conjunction)} \\
&\quad \mid \text{FuzzySet OR FuzzySet} && \text{(Disjunction)} \\
&\quad \mid \text{NOT FuzzySet} && \text{(Negation)} \\
&\quad \mid \text{LinguisticVariable} = \tilde{A} && \text{(Selection)}
\end{aligned}$$

The boolean operators AND, OR, and NOT represent the fuzzy set operations of intersection, union, and complement. The selection operator LinguisticVariable $= \tilde{A}$ states that we are interested in the fuzzy set $\tilde{A}$ part of the linguistic variable. It is not an assignment but rather acts as syntactic sugar to make the rules more readable. A typical rule could look like this:

$$\text{IF } (\textit{age} = \text{"young" AND } \textit{height} = \text{"tall"}) \text{ THEN fitness} = \text{"good"}$$

This rule states that if the state of the linguistic variable *age* is *"young"* and the state of the linguistic variable *height* is *"tall"*, then the state of the linguistic variable *fitness* should be *"good"*. The actual ranges of the fuzzy sets *"young"*, *"tall"*, and *"good"* are defined with the corresponding membership functions and are not part of the rule itself.

2.3.5. Fuzzy Inference

The inference step can be seen as an extension of the boolean implication operator

$$\text{IF antecedent THEN consequent} \iff (\text{antecedent} \implies \text{consequent})$$

Instead of deriving the membership function of the implication operator from the t-conorm and the negation operator as in classical logic, the Mamdani implication is typically used in fuzzy logic. This particular implication is defined as the minimum t-norm operation $\min(a, b)$. This choice is counterintuitive as it does not mimic its equivalent in classical logic;

however, in the context of fuzzy systems, it is a preferred choice, as instead of evaluating the truthiness of the whole implication, it computes the degree to which the consequent should be activated [BMK96].

Evaluation of Fuzzy Logic Rules

Consider the rule $IF\,(a = \tilde{A}\ AND\ b = \tilde{B})\ THEN\ c = \tilde{C}$. To calculate the result of this rule, we perform the following steps:

1. ***Fuzzification*:** Obtain the crisp input values $(x_1, x_2, \ldots, x_n) \in X_A$ occurring in the crisp set of the antecedent and evaluate the degree of membership μ_i of each fuzzy set $\tilde{A}_i$ contained in the antecedent for each fuzzy rule (Here: $\mu_{\tilde{A}}(x_1)$ and $\mu_{\tilde{B}}(x_2)$).
2. ***Rule Activation*:** Calculate the total degree of activation μ by combining all membership values with the appropriate fuzzy logic operators (Here: $\mu = \min(\mu_{\tilde{A}}(x_1), \mu_{\tilde{B}}(x_2))$).
3. ***Rule Evaluation*** Define a new fuzzy set $\tilde{C}_{new} = \tilde{C} \uparrow \mu$ as the result of the rule activation, where $\tilde{C}$ is the the consequent and $\uparrow$ is the cut operator. The cut operator is defined as $\mu_{\tilde{C}\uparrow\mu}(x) = \min(\mu_{\tilde{C}}(x), \mu)$ following the Mamdani implication.
4. ***Aggregation*:** Combine the resulting fuzzy sets of each rule acting on the same linguistic variable using the fuzzy union operator (maximum t-conorm). This ensures all rules are considered when determining the final output.
5. ***Defuzzification*:** Calculate the defuzzified value of the fuzzy set $\tilde{C}_{new}$ to obtain the crisp output value.

A visual depiction of these steps is shown in Figure 2.5.

Defuzzification

The final step in a Fuzzy Logic system is the defuzzification step. In this step, the resulting fuzzy created by the aggregation step is converted back into a crisp, numeric value that can be used as a concrete output or decision. There are different ways to defuzzify a fuzzy set, but a common theme is finding a single representative value that maintains certain aspects of the fuzzy set. In later sections, we will make use of the following two defuzzification methods:

- **Center of Gravity (COG)**
  The centroid method calculates the x-position of the center of mass of the fuzzy set for defuzzification. This method tries to find a weighted interpolation of all the activated fuzzy sets and tries to find an optimal compromise between all the possible terms. The centroid method is the most common defuzzification method in practice due to its simplicity and robustness. It considers every activated fuzzy set and thus makes full use of all available information. It is defined as:

$$\text{Center of Gravity}_x = \frac{\int_X x \cdot \mu_{\tilde{C}}(x)\,dx}{\int_X \mu_{\tilde{C}}(x)\,dx} \tag{2.9}$$

- **Mean of Maximum (MOM)**
  The Mean of Maximum method is simpler than the centroid method and only considers values, resulting in the highest possible membership value. If multiple such values exist, the arithmetic mean of those values is returned. Contrary to the centroid method, there is usually no interpolation between the different fuzzy sets, as they usually have different degrees of activation. Consequently, the result is often based on a single linguistic term with the highest membership value. It is defined as follows:

$$\text{Mean of Maximum} = \frac{\int_{X'} x\,dx}{\int_{X'} dx} \tag{2.10}$$

  where X' is the set of all x where $\mu_{\tilde{C}}(x)$ is maximal.

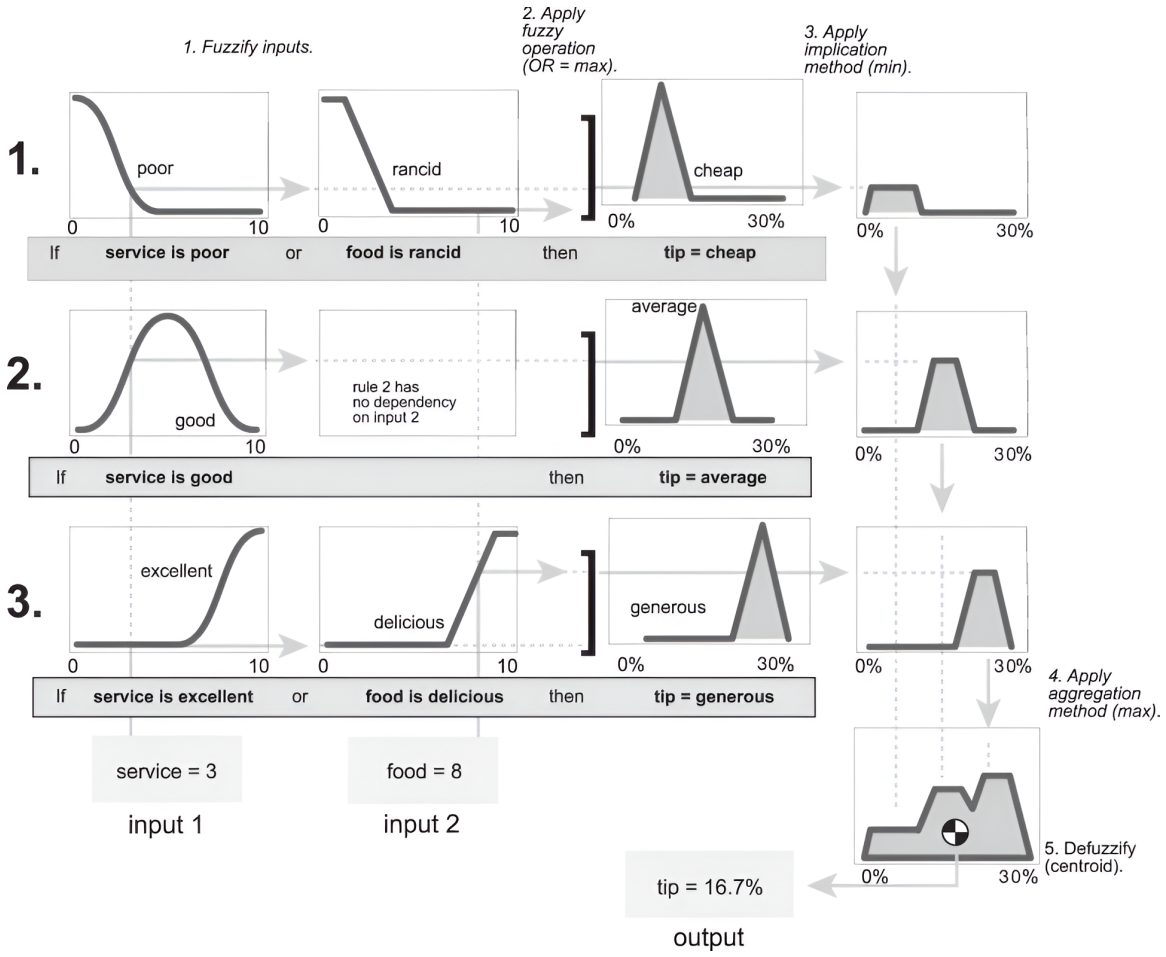


Figure 2.5.: Visualization of the full fuzzy logic inference process. The system consists of three fuzzy rules connecting the linguistic variables *service* and *food* to the linguistic variable *tip*. Applying each rule yields a resulting fuzzy set (depicted on the rightmost column for each rule), which is then aggregated using the fuzzy union operator to form the final output (depicted on the bottom right). This final fuzzy set is then defuzzified to obtain a crisp output value. Source: MathWorks - Fuzzy Inference Process

3. Implementation

This chapter describes the implementation of the Fuzzy Tuning technique in AutoPas. The implementation is divided into three main parts: the generic Fuzzy Logic Framework, the Rule Parser, and the Fuzzy Tuning Strategy. The Fuzzy Logic Framework is the core of this implementation and implements the mathematical foundation of this technique. The Rule Parser loads the supplied knowledge base from a rule file. Finally, the Fuzzy Tuning Strategy implements the interface between the Fuzzy Logic Framework and the AutoPas simulation.

3.1. Fuzzy Logic Framework

The Fuzzy Logic framework implements the mathematical foundation of the Fuzzy Tuning technique. It consists of several classes:

- **Crisp Set**
  The Crisp Set class models classical sets using k-cells[1] in order to represent the universe of discourse for the fuzzy sets. Therefore, it keeps track of the ranges of the input variables, which are later used in the defuzzification step. Using k-cells, we can only model continuous variables with a finite range of values. This is an acceptable limitation for the current use case in AutoPas, as all relevant parameters either fulfill this requirement or can be encoded as such (see Subsection 3.3.1). However, there exist methods to directly use nominal values as described in [RdCC12] or [JPRS06], but those are not implemented in this work.
- **Fuzzy Set**
  As mentioned previously, fuzzy sets consist of a membership function $\mu : C \to [0, 1]$, assigning a degree of membership to each element of the associated Crisp Set C. For the implementation in C++, we distinguish between two types of membership functions: The `BaseMembershipFunction` and the `CompositeMembershipFunction`. The `BaseMembershipFunction` implements membership functions over 1-dimensional k-cells (1-cells), in particular intervals in the real numbers $\mathbb{R}$. It represents *conventional* membership functions and is implemented as a lambda function $f : \mathbb{R} \to [0, 1]$ that directly assigns the degree of membership to each input value. Commonly used examples, such as triangular, trapezoidal, gaussian, and sigmoid-shaped membership functions, are implemented this way and can be selected by the user via the rule file.

  The `CompositeMembershipFunction` implements membership functions over higher dimensional k-cells. This distinction is necessary, as we will use a recursive approach to construct complex fuzzy sets from simpler ones, and those newly constructed fuzzy

[1] A k-cell is a hyperrectangle in the k-dimensional space constructed from the Cartesian product of k intervals $C = I_1 \times I_2 \times \ldots \times I_k$ where $I_i = [x_{low}, x_{high}] \subset \mathbb{R}$ is an interval in the real numbers.

sets should compose their children's membership functions to calculate their own membership value, thus requiring a different interface than the `BaseMembershipFunction`. The `CompositeMembershipFunctions` are automatically constructed when applying logical operations to fuzzy sets. To demonstrate the concept, let us consider the fuzzy set $\tilde{C} = \tilde{A} \cap \tilde{B}$. This new fuzzy set $\tilde{C}$ is defined over the Crisp Set $C = A \times B$, where A and B are the Crisp Sets of the fuzzy sets $\tilde{A}$ and $\tilde{B}$, respectively. As explained in previous chapters, the membership function $\mu_{\tilde{C}} : C \to [0, 1]$ can be calculated as $\mu_{\tilde{C}}(x, y) = \min(\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(y))$, thus recursively making use of the membership functions of the *child* fuzzy sets $\tilde{A}$ and $\tilde{B}$. The only new information the `CompositeMembershipFunction` needs to store is the function that should be used to combine the membership values of the children. As these membership functions need to provide multiple arguments to their child fuzzy sets, they are implemented as lambda functions $f : \mathbb{R}^k \to [0, 1]$. Complex fuzzy sets, resulting from logical operations, are implemented this way. The membership function primarily stores information on how to combine the membership values of the children (e.g., min, max, $1 - \cdot$).

Internally, all fuzzy sets are represented using a tree data structure. The tree's root node represents the fuzzy set itself, and every internal node represents a fuzzy set from a subexpression. In this tree structure, the `CompositeMembershipFunctions` act as a link between existing fuzzy sets (the *children*) and leads to the definition of a more complex fuzzy set (the *parent*). The Leaf nodes of a fuzzy set can no longer be decomposed into simpler fuzzy sets and are consequently defined using the `BaseMembershipFunctions`. Figure 3.1 shows a larger example of how complex fuzzy sets can be constructed from simpler fuzzy sets using this recursive approach. The Fuzzy Set class also provides methods for defuzzification and combining fuzzy sets using logical operations.

- **Linguistic Variable**
  Linguistic variables act as simple containers for fuzzy sets. Each Linguistic Variable has a name (e.g., `temperature`) and stores linguistic terms, each consisting of a name (e.g., `hot`) and a corresponding fuzzy set $\tilde{H}$ describing the *distribution* of the term.
- **Fuzzy Rule**
  The Fuzzy Rule class stores an antecedent and a consequent fuzzy set ($\tilde{A}$ and $\tilde{C}$). Additionally, the class provides a method to apply the rule, producing a new fuzzy set $\tilde{C}_{new} = \tilde{C} \uparrow \mu$ consisting of the partially activated fuzzy set $\tilde{C}$ where μ is the degree of membership of the supplied input values in the antecedent fuzzy set $\tilde{A}$.
- **Fuzzy System**
  The Fuzzy System class combines all the concepts described above to create arbitrary systems to evaluate a set of fuzzy rules and generate a final, defuzzified output value. Such a system acts like a black box $f : \mathbb{R}^n \to \mathbb{R}$ mapping crisp input values to crisp output values. Later sections make use of this class to implement the Fuzzy Tuning Strategy.

The full implementation at the time of writing can be found in the AutoPas repository at src/autopas/tuning/tuningStrategy/fuzzyTuning. A simplified class diagram of the Fuzzy Tuning Strategy and the Fuzzy Logic Framework can be seen in Figure 3.2.

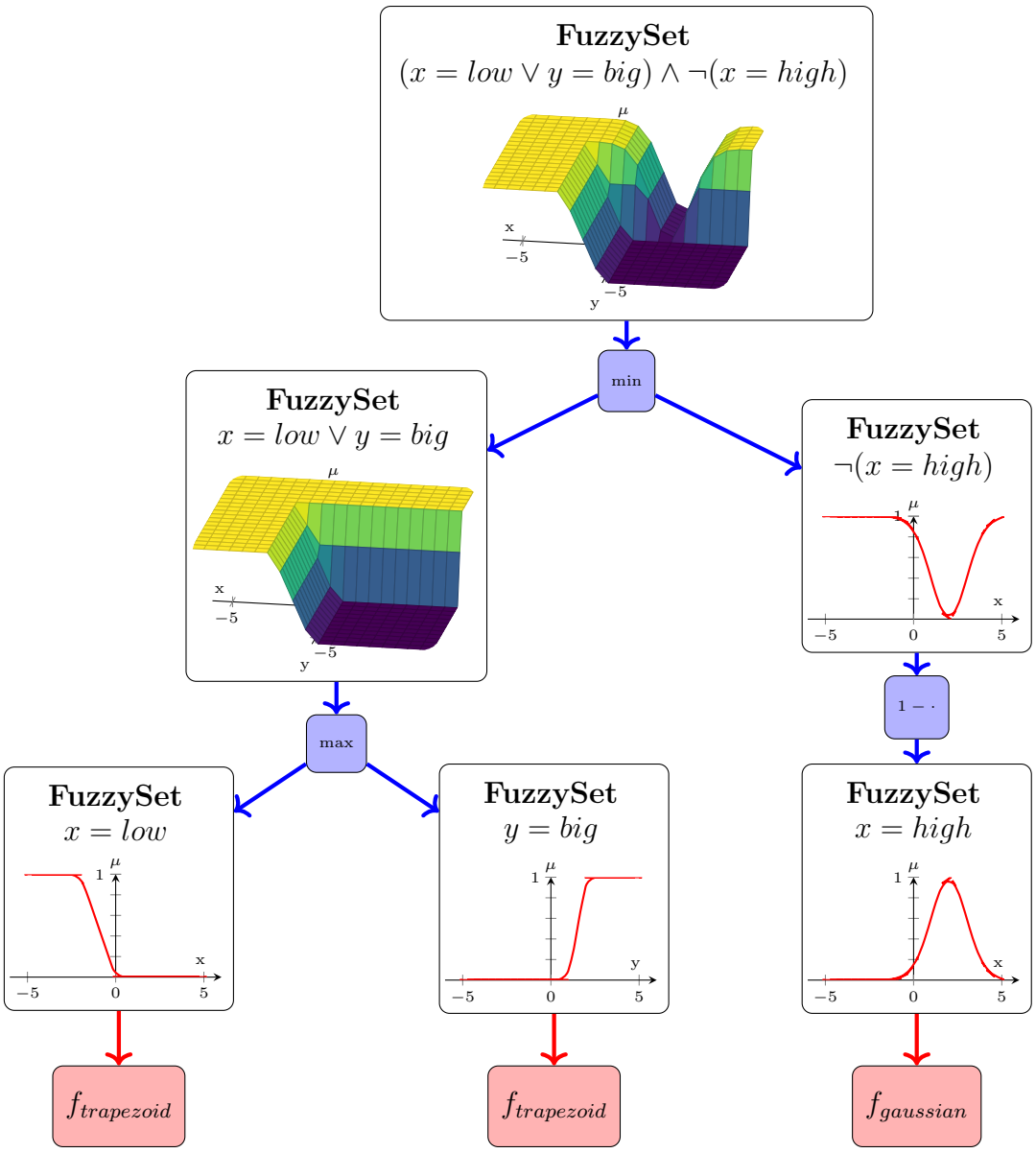


Figure 3.1.: Recursive construction of a complex fuzzy set from simpler fuzzy sets. Using the linguistic variables x with the fuzzy sets $\{low, high\}$ and y with the fuzzy sets $\{big, small\}$ we can construct the fuzzy set $(x = low \vee y = big) \wedge \neg(x = high)$ by combining simpler fuzzy sets as shown in the figure.
The fuzzy sets at the leaf level can be directly evaluated using predefined `BaseMembershipFunctions` (e.g., trapezoid, sigmoid, gaussian ...) and provide the foundation for the more complex fuzzy sets All other fuzzy sets are created by combining existing fuzzy sets using `CompositeMembershipFunctions`. The union operator between fuzzy sets corresponds to the max function, the intersection operator corresponds to the min function, and the negation operator corresponds to the $1 - \cdot$ function.

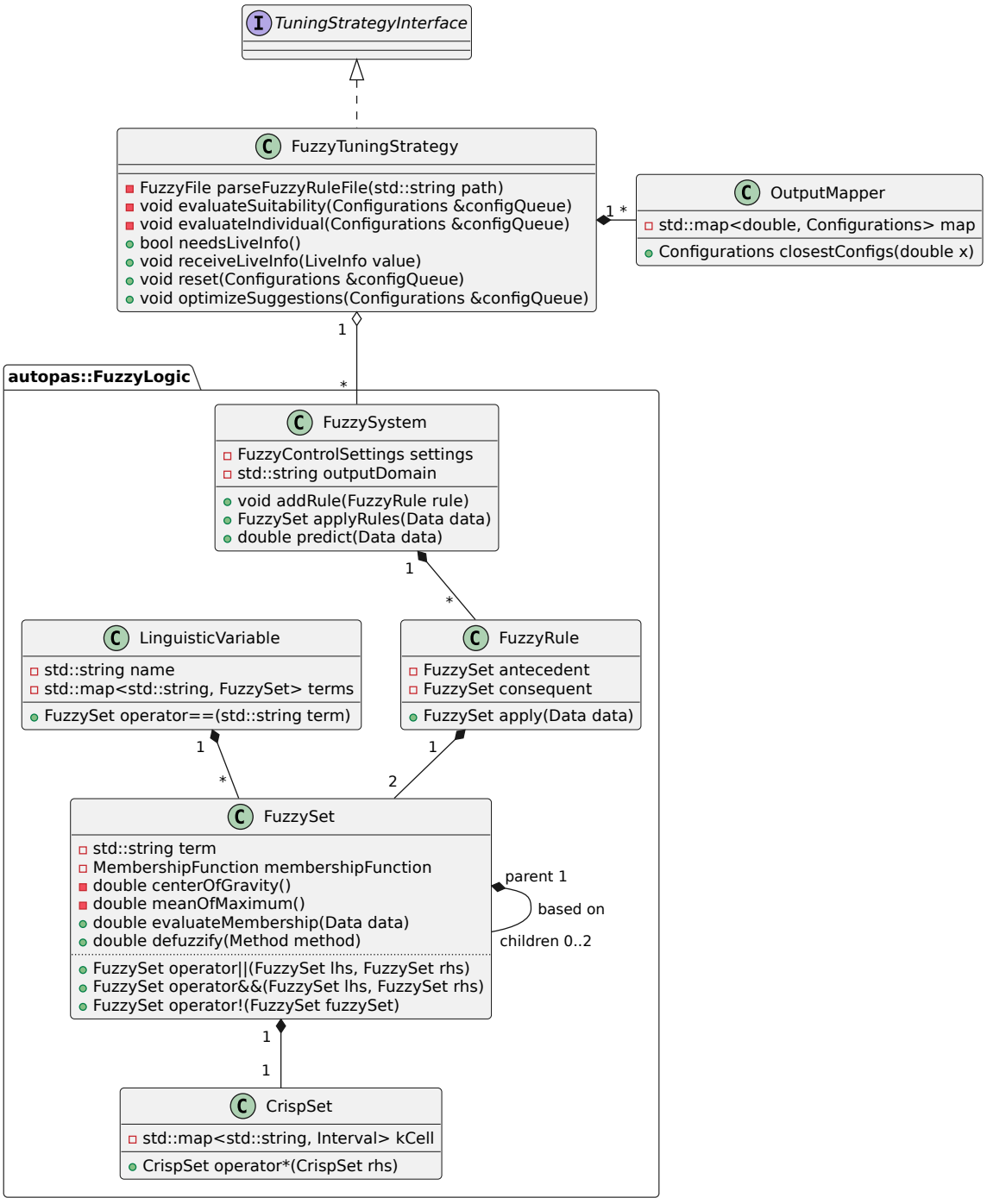


Figure 3.2.: Simplified class diagram of the Fuzzy Tuning strategy. There is a clear separation between implementing the Fuzzy Logic Framework and the tuning strategy. This allows for an easy reuse of the Fuzzy Logic Framework in other parts of AutoPas if desired.

3.2. Rule Parser

The Rule Parser is responsible for parsing the knowledge base supplied by the user and converting it into the internal representation used by the Fuzzy Logic Framework. It is based on the ANTLR4[2] parser generator and makes use of a domain-specific language tailored to the needs of the Fuzzy Tuning. The language is designed to be lightweight and directly incorporates aspects of AutoPas, such as configurations, into the rule file. All supplied rules predicting values for the same output variable are grouped to form a single Fuzzy System responsible for this output variable.
The conversion between the generated parse tree and the internal representation is done by a visitor pattern that traverses the parse tree generated by ANTLR4 and internally builds the corresponding object hierarchy. A small example demonstrating the syntax of the rule file can be seen in Listing 3.1.

```
# Define the settings of the fuzzy systems
FuzzySystemSettings:
    defuzzificationMethod:  "meanOfMaximum"
    interpretOutputAs:      "IndividualSystems"

# Define linguistic variables and their linguistic terms
FuzzyVariable: domain: "homogeneity" range: (-0.009, 0.1486)
    "lower than 0.041":     SigmoidFinite(0.0834, 0.041, -0.001)
    "higher than 0.041":    SigmoidFinite(-0.001, 0.041, 0.0834)

FuzzyVariable: domain: "threadCount" range: (-19.938, 48.938)
    "lower than 18.0":      SigmoidFinite(38.938, 18.0,  -2.938)
    "lower than 26.0":      SigmoidFinite(46.938, 26.0,   5.061)
    "lower than 8.0":       SigmoidFinite(28.938,  8.0, -12.938)
    "higher than 18.0":     SigmoidFinite(-2.938, 18.0,  38.938)
    "higher than 26.0":     SigmoidFinite(5.0617, 26.0,  46.938)
    "higher than 8.0":      SigmoidFinite(-12.93,  8.0,  28.938)

FuzzyVariable: domain: "particlesPerCellStdDev" range: (-0.017, 0.072)
    "lower than 0.013":     SigmoidFinite(0.0639, 0.038,  0.012)
    "higher than 0.013":    SigmoidFinite(0.012,  0.013,  0.0639)

FuzzyVariable: domain: "Newton 3" range: (0, 1)
    "disabled, enabled":    Gaussian(0.3333, 0.1667)
    "enabled":              Gaussian(0.6667, 0.1667)

# Define how the output variables should be decoded into configurations of AutoPas
OutputMapping:
 "Newton 3":
     0.3333 => [newton3 = "disabled"], [newton3 = "enabled"]
     0.6667 => [newton3 = "enabled"]

# Define rules connecting the input variables to the output variables
if ("threadCount" == "lower than 18.0") && ("threadCount" == "higher than 8.0")
     && ("homogeneity" == "lower than 0.041")
   then ("Newton 3" == "enabled")

if ("threadCount" == "higher than 26.0") && ("particlesPerCellStdDev" == "lower than
     0.013")
   then ("Newton 3" == "disabled, enabled")
```

Listing 3.1: Demonstration of the domain-specific language used for Fuzzy Tuning

[2]https://www.antlr.org/

3.3. Fuzzy Tuning Strategy

The Fuzzy Tuning Strategy implements the interface between the Fuzzy Logic framework and the AutoPas simulation and is responsible for updating the configuration queue of configurations to be tested next. To achieve this, the strategy evaluates all fuzzy systems present in the rule file using the *LiveInfoData* (See A.2) collected by AutoPas. These data points contain summary statistics about various aspects of the current simulation state, such as the total number of particles, the average particle density, or the average homogeneity of the particle distribution. The fuzzy systems should use those values to calculate the results. Each evaluation of a Fuzzy System yields a single numeric value, which is then passed on to the `OutputMapper` object. The `OutputMapper` is responsible for mapping the continuous output value of the Fuzzy System to the discrete configuration space of AutoPas.

Internally, the `OutputMapper` stores an ideal numerical location for each configuration-pattern[3] and always selects the option closest to the predicted value. This method of assigning discrete values to the output of fuzzy systems is inspired by Mohammed et al.'s [MKEC22] work on scheduling algorithms, where the authors used a similar approach.

All the configuration patterns predicted by the Fuzzy Systems are then collected and used to update AutoPas's configuration queue of configurations to be tested next during the tuning phase.

Currently, two different approaches using Fuzzy Tuning to predict *optimal* configurations are implemented: The *Component Tuning Approach* and the *Suitability Tuning Approach*. Both approaches are described in detail in the following sections.

3.3.1. Component Tuning Approach

The Component Tuning Approach assumes that each tunable parameter can be tuned independently of the others, making it possible to define a separate Fuzzy System for each tunable parameter.

All those Fuzzy Systems should then attempt to predict the best value of their parameter independent of the other parameters. This approach requires the rule file to only define $\#Parameters$ different Fuzzy Systems and a corresponding `OutputMapper` for each parameter. Creating such rule files is straightforward and could be reasonably created manually by a domain expert. An obvious drawback of this method is the independence assumption between the parameters, which might not hold in practice. However, the practical Experiments carried out in Chapter 5 still show quite good results, even with this simplification.

Another problem of this approach lies in the defuzzification step. As this method relies on defining a single system for all values of a tunable parameter, we must define a numerical *ranking* of all values the parameter could take. Such a ranking is problematic, as most tunable variables are nominal and thus do not have a natural order (e.g., lc_c04, lc_c08, vcl_c06, vcl_sliced_balanced ...). To circumvent this problem, we chose the MOM defuzzification method, which selects the mean of all x-values for which the membership function is maximal.

[3] A configuration-pattern is a tuple of all tunable parameters, where each component of the tuple describes a set of possible values for this parameter. The wildcard value `*` allows any possible value. For example, the configuration-pattern `(Container=LinkedCells, Traversal=*, DataLayout=SoA, Newton3=enabled)` matches the specified configuration, regardless of the value of the `Traversal` parameter.

When using Gaussian-shaped membership functions for the output values, this method will always return the mean of the Gaussian with the highest activation[4].

After evaluating this ruleset, one ends up with a list of configuration patterns, each describing a different pattern to which the solution should conform. All those patterns are then used to filter the configuration queue, excluding every configuration that does not match all the predicted patterns. Figure 3.3 shows a schematic of how the Fuzzy Tuning Strategy could be used for the Component Tuning Approach.

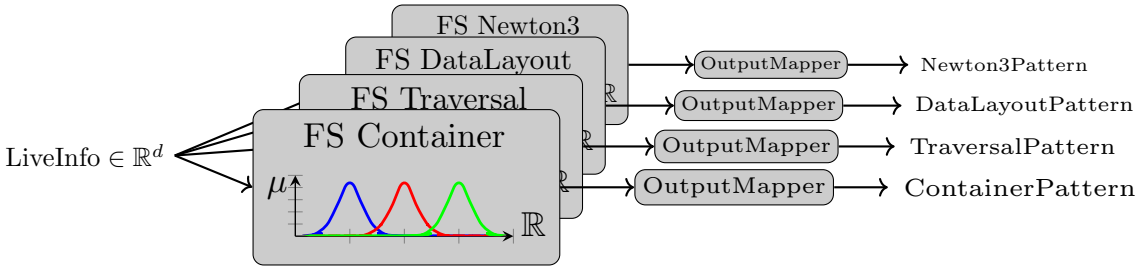


Figure 3.3.: Example Visualization of the fuzzy systems for the Component Tuning Approach. The parameters `Container`, `Traversal`, `DataLayout`, and `Newton3` are tuned independently. The OutputMapper maps the defuzzified output values to their corresponding configuration patterns. The configuration queue is then updated with all configurations that match all predicted patterns.

3.3.2. Suitability Tuning Approach

The Suitability Approach mainly differs from the Component Tuning Approach in that it utilizes $\#Container\_options \cdot \#Traversal\_options \cdot \#DataLayout\_options \cdot \#Newton3\_options$ different Fuzzy Systems, one for each possible combination of those parameters. Each Fuzzy System is responsible for predicting the suitability of its configuration.

The advantage of this approach is that there is no need to rank the output values, and one can utilize the power of Fuzzy Systems to interpolate between different predictions. This method uses the center of gravity (COG) defuzzification method, as suitability values have a natural order (higher suitability is better). Furthermore, dependencies and incompatibilities between the parameters can be modeled accurately, as each way of combining the parameters is handled with a separate Fuzzy System. The downside of this method is the enormous complexity of the rule file, which quickly becomes infeasible to maintain by hand. Surprisingly, the cost of evaluating all those Fuzzy Systems is negligible compared to the overhead of other tuning strategies, as later experiments in Chapter 5 will show.

After evaluating all Fuzzy Systems and using a trivial OutputMapping, the method yields a list of `(Configuration, Suitability)` pairs, which can then be used to update the configuration queue. The current implementation selects the highest possible suitability value and then chooses every configuration performing within a certain threshold of the best configuration. Those configurations are then used to overwrite the configuration queue.

[4]There are exceptions when two Gaussians have the same level of activation, in which case the mean of both Gaussians is returned. However, this rarely happens in practice and could be resolved with special defuzzification methods. The current implementation just uses the MOM method as it works well in practice.

Figure 3.4 shows how the Fuzzy Tuning Strategy could be used for the Suitability Tuning Approach.

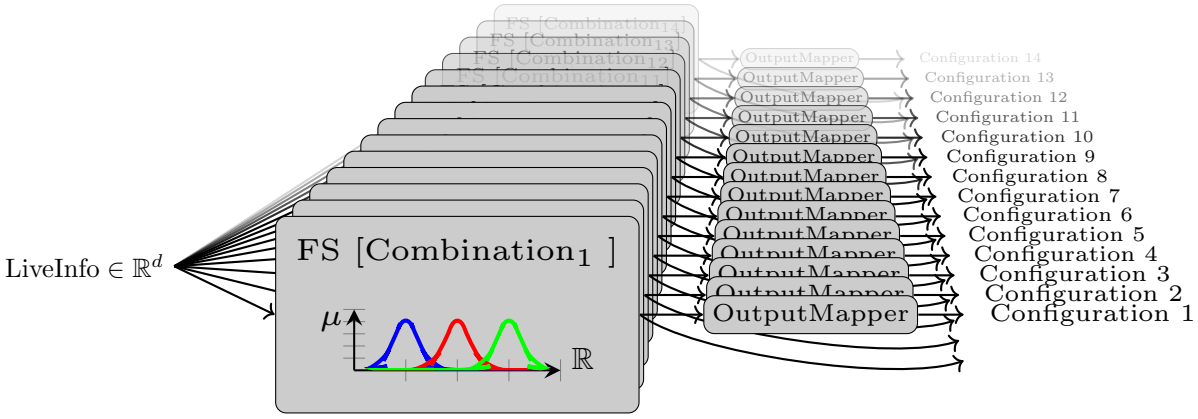


Figure 3.4.: Example Visualization of the fuzzy systems for the Suitability Tuning Approach. Each fuzzy system is responsible for predicting the suitability of a specific combination of tunable values, resulting in an enormous amount of fuzzy systems. The `(Configuration, Suitability)` pairs are passed to the Fuzzy Tuning Strategy, which then updates the configuration queue based on the highest suitability values.

4. Proof of Concept

This chapter presents a proof of concept for the fuzzy tuning technique and will develop working instantiations of the approaches introduced in the previous chapter.

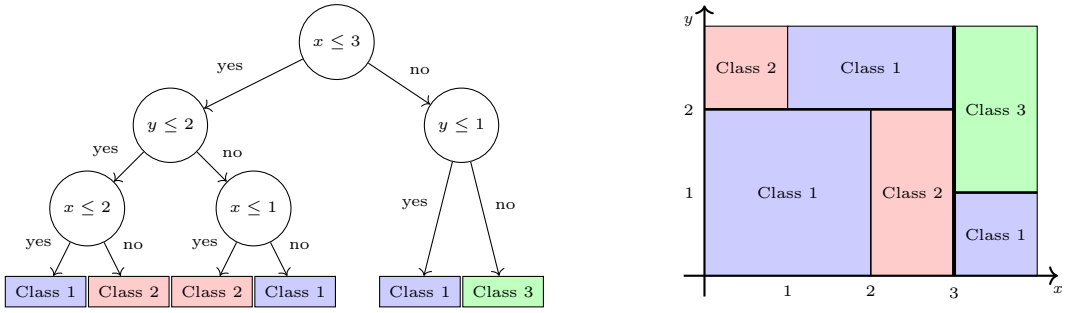
Creating rule bases for fuzzy systems is challenging, as it typically requires a profound prior understanding of the system to create meaningful rules. In practice, such knowledge may be difficult to acquire and formalize, as experts may struggle to express their intuition and experience in the precise, structured form required for a fuzzy rule base. However, data-driven approaches can help semi-automate this process by generating an initial set of rules without prior expert knowledge, which can be refined later.

In this work, we will use a data-driven approach based on decision trees proposed by Crockett et al. [CBMO06]. This method first trains decision trees to create an initial set of crisp rules. In the second step, the decision trees are converted into so-called fuzzy decision trees, which can then be used to extract the linguistic variables and fuzzy rules.

4.1. Data Driven Rule Extraction

4.1.1. Decision Trees

Decision trees are prevalent machine learning algorithms used for classification and regression tasks. They work by recursively partitioning the input using axis-parallel splits so that the resulting subsets are as pure as possible. In particular, they try to minimize a given impurity metric, such as the Gini impurity $I_G = \sum_{i=1}^{n} p_i(1 - p_i)$ of the subsets [Mur12]. Since decision trees directly partition the input space into regions with different classes, they can also be depicted using their decision surface if the dimensionality allows it. The decision surface of a decision tree is a piecewise constant function that assigns the predicted class label to each point in the input space of the decision tree. An example decision tree and its decision surface are shown in Figure 4.1a and Figure 4.1b, respectively.



(a) Example decision tree

(b) Decision surface over $\mathcal{D} = [0, 4] \times [0, 3]$

Figure 4.1.: Example decision tree and its decision surface

4.1.2. Conversion of Decision Trees to Fuzzy Systems

This section will demonstrate how to convert a classical decision tree into a fuzzy decision system using the fictional decision tree from Figure 4.1a as an example.

Fuzzy Decision Trees

Fuzzy decision trees are a generalization of classical decision trees and allow for fuzzy logic to be used in the decision-making process. Instead of following the classical `if then else` logic to descend into the decision tree, it uses fuzzy sets at each node of the tree to fuzzily calculate the contribution of each branch to the final decision based on the degree of truth of both possible paths. Contrary to classical decision trees, which follow a single path from the root to a leaf node, fuzzy decision trees explore all possible paths simultaneously and make a final decision by aggregating the results of the paths using fuzzy logic operations.

Conversion

A classical decision tree is converted into a fuzzy decision tree by replacing the crisp decision (e.g., $x \leq 3$) at each internal node of the decision tree with fuzzy sets. Those fuzzy sets should maintain the same semantics as the crisp decision but should provide a continuous value in the range $[0, 1]$ specifying the degree of how much each branch should be considered. Allowing the decision trees to consider multiple paths simultaneously can drastically increase the decision-making capabilities of the decision tree, especially in boundary cases where the decision can be ambiguous.

The shape of the membership functions of the fuzzy sets can be chosen arbitrarily, but typical choices include complementary `sigmoid`-shaped functions that are centered around the crisp decision boundary (See Figure 4.2), as those function shapes maintain the semantics of the branching idea. Crockett et al. [CBMO06] suggested creating those sigmoid shapes with a *width* proportional to the standard deviation of the attribute. In Particular, the authors suggested an interval $[t - n \cdot \sigma, t + n \cdot \sigma]$ for the membership function, where t is the value of the decision boundary, σ is the standard deviation of the attribute and n is a parameter that can be adjusted to control the *width* of the membership function. This interval specifies the region of the membership function where most of the change in membership occurs. The value of n is typically chosen from the interval $n \in [0, 5]$ as the membership function can become too broad otherwise and could even weaken the decision-making process [CBMO06]. In this work, we will use $n = 2$ as a default value.

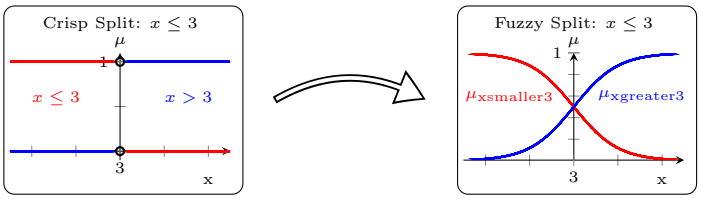


Figure 4.2.: Conversion of crisp membership functions to fuzzy membership functions. The crisp membership functions $x \leq 3$ and $x > 3$ of a decision tree node are replaced by two `sigmoid`-shaped membership functions $\mu_{\text{xsmaller3}}$ and $\mu_{\text{xgreater3}}$.

Once the internal nodes of the decision tree have been converted, the next step is to convert the leaf nodes of the decision tree to fuzzy leaf nodes. As the outputs of decision trees are specific class labels, we can define a single linguistic variable consisting of all possible class labels each associated with a fuzzy set. The shapes of the membership functions for the base fuzzy sets can again be chosen mostly arbitrarily, but we will use `gaussian` functions with a different mean as they are a good choice for representing class labels in a continuous domain. The resulting conversion of the decision tree in Figure 4.1a into a fuzzy decision tree is shown in Figure 4.3.

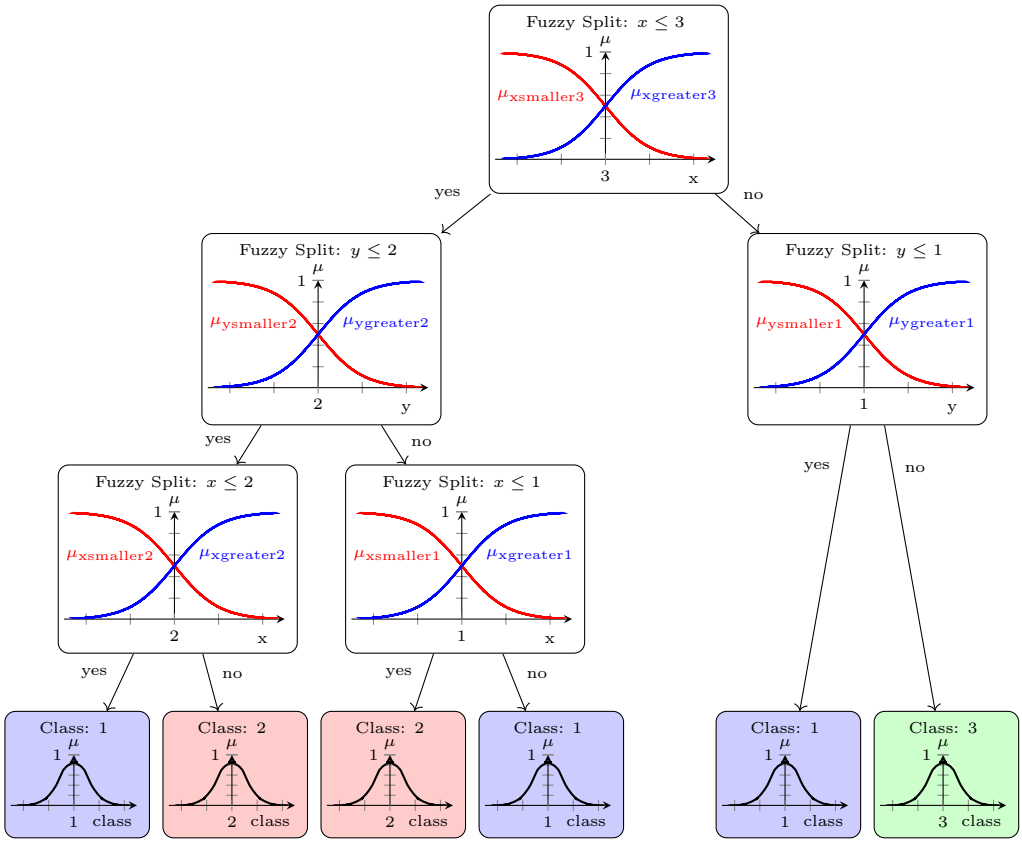


Figure 4.3.: The fuzzy decision tree corresponding to the decision tree in Figure 4.1a. Internal nodes use two `sigmoid` membership functions (μ_{smaller} and μ_{greater}) instead of a crisp decision. The leaf nodes use different `gaussian` membership functions centered around a unique mean.

It is now possible to fully extract all linguistic variables from the fuzzy decision tree. Every fuzzy set in the tree can be grouped into a corresponding linguistic variable based on the input variable of the fuzzy set. This results in linguistic variables consisting of a bunch of different `sigmoid` membership functions for input variables (internal nodes) and a single linguistic variable with different `gaussian` membership functions for the output variable (leaf nodes). The resulting linguistic variables are shown in Figure 4.4.

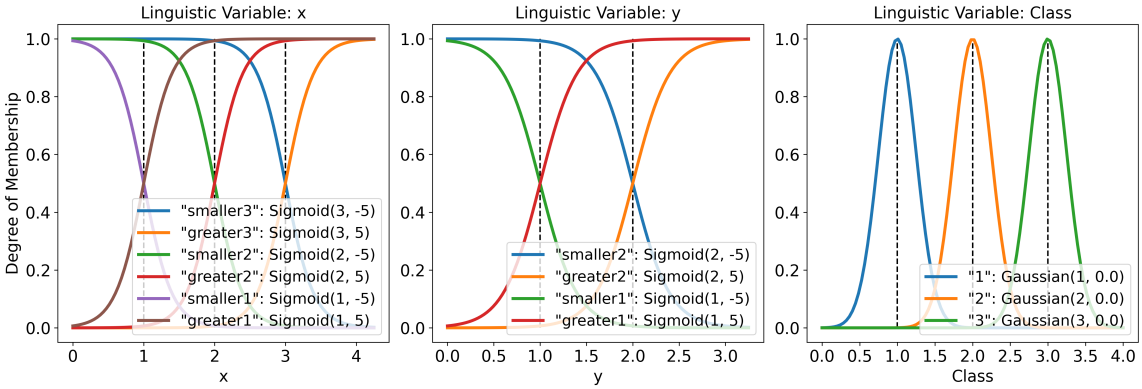


Figure 4.4.: Linguistic variables used in the fuzzy decision tree of Figure 4.3. The standard deviation of the attributes is assumed to be $\sigma \approx 0.5$ such that the *width* of the sigmoid membership functions is $n \cdot \sigma \approx 1$. The standard deviation and placement of the class values are chosen so that they do not overlap too much.

Rule Extraction

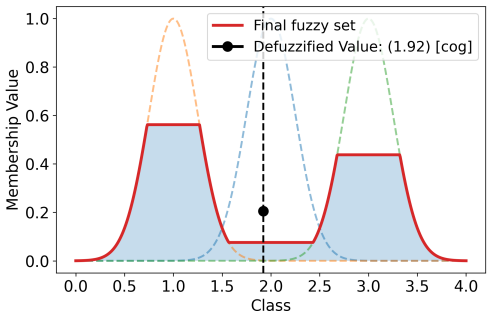
The final step is to extract the fuzzy rules from the tree. This can be done by traversing and aggregating the tree in a depth-first manner and collecting the correct membership functions for each path ending in a leaf node along the way. As all conditions of this path have to hold simultaneously, all fuzzy sets are connected using the fuzzy `AND` operation. This newly created fuzzy set represents the antecedent of the rule. The fuzzy set representing the class (the leaf node) is the consequent of the rule. Therefore, each unique path traversing the tree results in a unique rule of the form *IF antecedent THEN consequent*. This process essentially mimics the decision surface seen in Figure 4.1b, as we create precisely one rule for each region of the decision surface. The rules extracted from the fuzzy decision tree in Figure 4.3 using this method are shown in Table 4.1.

Rule	**Antecedent**	**Consequent**
1	x *is* smaller3 $\wedge$ y *is* smaller2 $\wedge$ x *is* smaller2	*class is* 1
2	x *is* smaller3 $\wedge$ y *is* smaller2 $\wedge$ x *is* greater2	*class is* 2
3	x *is* smaller3 $\wedge$ y *is* greater2 $\wedge$ x *is* smaller1	*class is* 2
4	x *is* smaller3 $\wedge$ y *is* greater2 $\wedge$ x *is* greater1	*class is* 1
5	x *is* greater3 $\wedge$ y *is* smaller1	*class is* 1
6	x *is* greater3 $\wedge$ y *is* greater1	*class is* 3

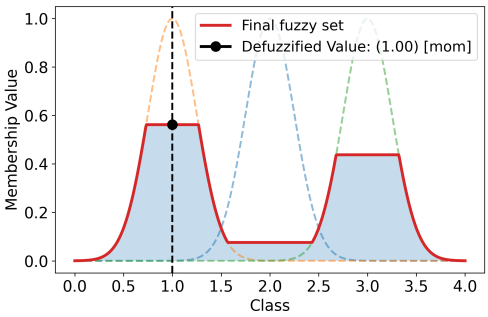
Table 4.1.: Extracted fuzzy rules from the fuzzy decision tree in Figure 4.3 in the format: **IF** Antecedent **THEN** Consequent

Effect of an Unsuitable Defuzzification Method

This section briefly discusses the effect of an unsuitable defuzzification method on the resulting fuzzy system. To show the problem when defuzzifying nominal variables, we use the previously extracted rules from Table 4.1 and evaluate the resulting Fuzzy Set for a critical data point $(x = 2.95, y = 2.5)$ using both the Center of Gravity (COG) and the Mean of Maximum (MOM) defuzzification methods. The resulting fuzzy sets are shown in Figure 4.5a and Figure 4.5b.



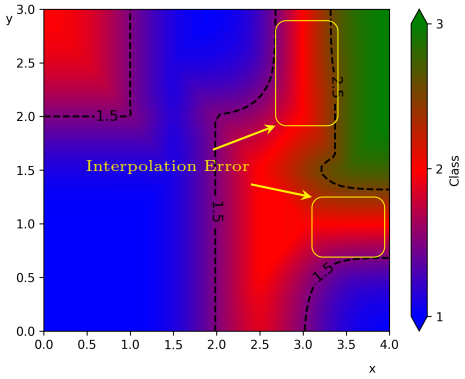
(a) Defuzzification using the COG method. Due to the interpolation, the COG method incorrectly suggests results in a class value close to 2.0 without it being a prominent class.



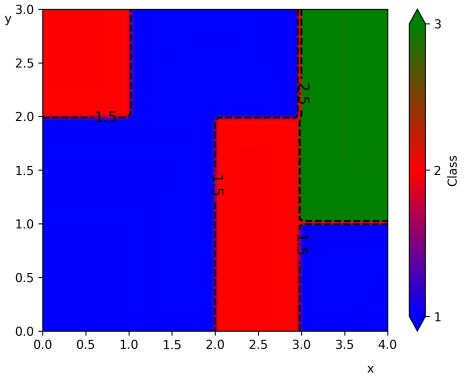
(b) Defuzzification using the MOM method. The MOM method correctly suggests the class value 1.0 as it is the most prominent class in the fuzzy set.

Figure 4.5.: Comparison of COG and MOM on data point $(x, y) = (2.95, 2.5)$.

To get a full understanding of this problem, we create the full decision surfaces for both fuzzy systems shown in Figure 4.6a and Figure 4.6b, respectively. The decision surface using COG tries to smoothly interpolate between the different classes, which causes interpolation errors if there are other classes in between. The decision surface using MOM is mostly valid and closely resembles the decision surface of the crisp decision tree in Figure 4.1b. Consequently, we should use the MOM method when working with nominal variables.



(a) Decision surface when using the COG method. The highlighted areas show interpolation errors.



(b) Decision surface when using the MOM method. There are only minor invalid regions in the decision surface.

Figure 4.6.: Comparison of COG and MOM decision surfaces of the fuzzy rules.

4.2. Fuzzy Systems for `md_flexible`

This section will use the data-driven approach introduced in the previous section to demonstrate how to generate fuzzy systems for `md_flexible` simulations. The first section will describe the data collection process needed to train the classic decision trees, and the later sections will use the obtained data to generate the two different styles of fuzzy systems introduced in Section 3.3.

4.2.1. Data Collection

Included Scenarios

We chose to include the prominent example scenarios provided by `md_flexible` such as `fallingDrop.yaml`, `explodingLiquid.yaml` and `SpinodalDecomposition.yaml` as the primary source of data. Additionally, we included some simulations of uniform cubes with different densities and particle counts to gather more data about the performance of the different configurations under lab-like conditions.

All simulations were run on the serial partition of the CoolMUC-2 cluster[1] and were repeated twice to account for fluctuations in performance. Furthermore, every simulation was repeated with 1, 4, 12, 24, and 28 threads to additionally gather data on how parallelization affects the ideal configuration.

All the values were collected with the newly created `PAUSE_SIMULATION_DURING_TUNING` CMake option to ensure that the simulation state does not change during the tuning phases. This guarantees a fair comparison of the tested configurations, as all of them are evaluated under the exact same conditions. To gather the maximal amount of data, we used the `FullSeach` tuning strategy, which executes all possible configurations during the tuning phases.

Collected Parameters

The existing `TuningDataLogger` and the newly created `LiveInfoLogger` classes of the AutoPas framework allow us to collect a wide variety of parameters during the simulation, such as the used configuration, information about the state of the simulation, and the measured timing data for each iteration of the simulation.

The complete shape of the collected data can be found in Section A.1 and Section A.2, respectively. We will however only make use of a subset of the available LiveInfo data, as we are only interested in *relative* values that do not change when the simulation is scaled up or down and are therefore primarily interested in: `avgParticlesPerCell`, `maxParticlesPerCell`, `homogeneity`, `maxDensity`, `particlesPerCellStdDev` and `threadCount`. This focus should help the fuzzy systems generalize better to unseen data, as they are less likely to overfit the training data.

Limitations

As the performance of machine learning models may degrade quickly when confronted with significantly different data than the data they were trained on, it is essential to collect a wide variety of scenarios to cover as many possible use cases as possible. As we only included a

limited number of scenarios, we have to keep in mind that the generated fuzzy systems will only be able to make confident predictions about scenarios similar to the included ones, and we should not expect them to generalize well to unseen data. To guarantee a fair evaluation of this tuning approach, we will only focus on slight variations of the included scenarios during the later evaluation phase in Chapter 5.

4.2.2. Data Preprocessing

In order to make predictions about the performance of different configurations, we first need to define an appropriate metric to compare them. As we *paused* the simulation during the tuning process with the `PAUSE_SIMULATION_DURING_TUNING` option, we can safely use the runtimes of the different configurations to compare them. Those runtimes are, however, absolute values and may differ significantly between tuning phases as the underlying simulation changes. To compare runtimes between different tuning phases, we introduce the concept of *relative speed*, which measures how well a configuration performs compared to the best configuration in the same tuning phase, and augment the collected timing data with this metric. The relative speed is calculated as

$$\text{relative speed}^{(i)}_{\text{config}} = \frac{t^{(i)}_{\text{best}}}{t^{(i)}_{\text{config}}} \tag{4.1}$$

Where $t^{(i)}_{\text{best}}$ is the runtime of the best configuration during the i-th tuning phase and $t^{(i)}_{\text{config}}$ is the runtime of the configuration we are interested in.

This value will range from 0 (being infinitely worse than the best configuration) to 1 (being equally good as the best configuration) for each configuration. Additionally, we chose to combine the fields `Container` and `DataLayout` of the configuration into a single field `ContainerDataLayout` as they are closely related and should be tuned together.

Table 4.2 shows the augmented dataset for creating the fuzzy systems. The next sections sections will describe how this dataset can be used to create fuzzy systems for the so-called *Component Tuning Approach* and the *Suitability Tuning Approach*.

ParticlesPerCell			Miscellaneous			Configuration			
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3	Relative Speed
0.905	23	0.0129	0.0354	0.531	1	LinkedCells_AoS	lc_sliced	enabled	0.450641
2.201	13	0.0144	0.0861	0.627	24	VerletListsCells_AoS	vlc_sliced	disabled	0.594117
0.905	18	0.0136	0.0431	0.319	4	LinkedCells_AoS	lc_sliced_c02	enabled	0.454632
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.2.: Augmented dataset used for creating the fuzzy systems. The dataset contains all collected parameters and the relative speed of the configuration compared to the best configuration in the same tuning phase.

4.2.3. Component Tuning Approach

This approach makes use of three different `FuzzySystems`, one for each of the tunable parameters of the simulation (`ContainerDataLayout`, `Traversal`, and `Newton3`). All those systems should independently predict the best configuration for their respective parameter based on the current LiveInfoData. Figure 3.3 shows the structure of this approach.

As we only want to create a fuzzy system predicting good-performing configurations, we naively remove all configurations performing worse than a certain suitability threshold (we chose 70%) as depicted in Figure 4.7. The remaining configurations, which are known to perform well, are then used to create the fuzzy systems.

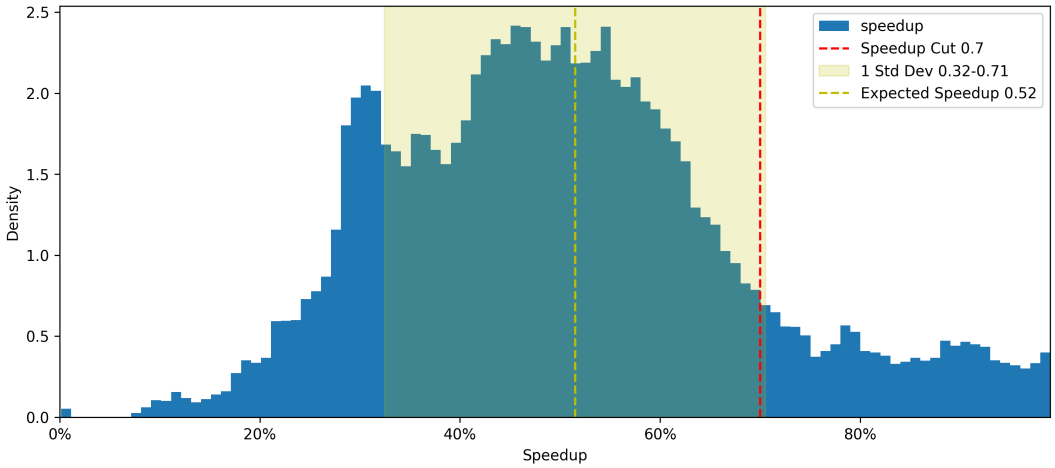


Figure 4.7.: Relative speed distribution of the collected data. The suitability threshold is set to 70%, thus removing all configurations performing worse than this threshold. From the plot, we can also see that the average configuration performs just 52% as well as the best configuration, with some configurations also performing ten times worse than the best in certain tuning phases.

Afterward, we group all configurations evaluated in the same tuning phase and aggregate all the present values of tunable parameters into a single term. As we *paused* the simulation during the tuning phase, the LiveInfoData will be equal for such configurations, and the aggregated terms will therefore represent all *good* values for the parameters in this simulation state (as they occur in configurations with $\geq 70\%$ suitability). The aggregated training data is shown in Table 4.3 and is used to fit three decision trees, which are then converted into fuzzy systems using the method described in the previous section. A few of the extracted fuzzy rules are shown in Table 4.4.

ParticlesPerCell			Miscellaneous			Aggregated Configuration Terms		
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3
0.906	15	0.015	0.055	0.297	4	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"	"lc_sliced, lc_sliced_balanced, lc_sliced_c02"	"enabled"
0.945	25	0.041	0.084	0.673	24	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"	"lc_c04, lc_c08, lc_sliced, lc_sliced_balanced"	"disabled, enabled"
0.906	20	0.014	0.041	0.336	24	VerletClusterLists_SoA, VerletListsCells_AoS	"vcl_c06, vlc_c01, vlc_c18"	"disabled, enabled"
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.3.: Aggregated training data for the Component Tuning Approach. Each row represents a different tuning phase. The numerical values stem from the LiveInfoData during that tuning phase, and the aggregated configuration terms represent the configuration options that are known to perform well under the given conditions.

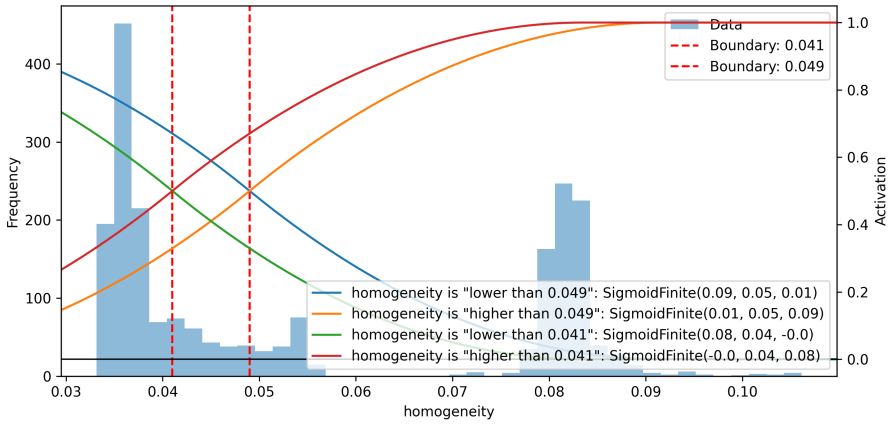
Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	ContainerDataLayout
`lower than 3.45`	`lower than 0.05`		`lower than 18.0`	"VerletClusterLists_SoA, VerletListsCells_AoS"
`lower than 3.45`	`higher than 0.05`	`lower than 0.024`	`higher than 18.0`	"LinkedCells_SoA, VerletClusterLists_SoA, VerletListsCells_AoS"
⋮	⋮	⋮	⋮	⋮

Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	Traversal
`lower than 1.553`	`higher than 0.047`	`lower than 0.023`	`higher than 2.5`	"lc_sliced, vlc_c18, lc_sliced_c02"
	`lower than 0.037`	`lower than 0.023`	`lower than 26.0`	"vcl_c06, vlc_c18, vlc_sliced_c02"
⋮	⋮	⋮	⋮	⋮

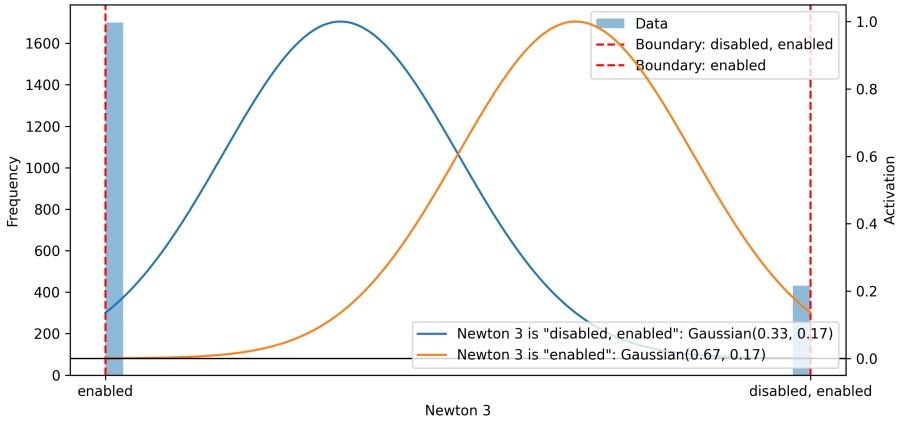
Antecedent				Consequent
avgParticlesPC	homogeneity	particlesPCStdDev	threadCount	Newton 3
		`higher than 0.03`	`higher than 18.0`	"disabled, enabled"
		`higher than 0.023` ∧ `lower than 0.037`	`lower than 18.0` ∧ `higher than 8.0`	"enabled"
⋮	⋮	⋮	⋮	⋮

Table 4.4.: Extracted fuzzy rules from the decision trees for the Component Tuning Approach. The rules are grouped by the tunable parameter they predict. The first row is read as: IF (avgParticlesPC = "lower than 3.454") ∧ (homogeneity = "lower than 0.05") ∧ (threadCount = "lower than 18.0") THEN (ContainerDataLayout = "VerletClusterLists_SoA, VerletListsCells_AoS")

As described previously, we use `gaussian` membership functions for each linguistic term of the consequent linguistic variables. The exact placement of the values is irrelevant as we will use the MOM defuzzification method, but they should be chosen so that they do not overlap completely. Figure 4.8a and Figure 4.8b show the resulting linguistic variables for the homogeneity linguistic variable (an input variable) and the Newton3 linguistic variable (an output variable). The visualizations of the other variables follow a similar pattern but are more complex due to the higher number of terms and are therefore not shown here.



(a) Linguistic variable for the Homogeneity attribute



(b) Linguistic variable for the Newton3 attribute

Figure 4.8.: Linguistic variables for Homogeneity and Newton3 attributes. The background shows the histogram values present in the dataset.

These linguistic variables and fuzzy rules are then used to create the fuzzy systems for the component tuning approach. After creating a suitable OutputMapper, we construct the final rule file for the component tuning approach, which can be looked up at `fuzzyRulesComponents.frule`.

4.2.4. Suitability Tuning Approach

The suitability approach differs from the component tuning approach in that it tries to predict a configuration's numerical *suitability* value under the current conditions. Therefore, each possible configuration is assigned a unique fuzzy system tailored to evaluate the suitability of its assigned configuration. Figure 3.4 shows the structure of this approach.

To train the decision trees, we again use a classification-based approach with the terms `terrible`, `poor`, `bad`, `medium`, `ok`, `good`, and `excellent` each corresponding to specific ranges of suitability values (see Figure 4.9 for the exact placement). Conveniently, we can use *suitability* = *relative speed*, as the relative speed value already measures how well a configuration performs.

We create the training data for the decision trees by adding a new column to the aggregated training data containing the suitability class to which the numeric relative speed value mostly belongs. The final training data is shown in Table 4.5. After grouping the data by possible configurations, it is again possible to use these data points to train the decision trees and extract the fuzzy rules from them. Due to the grouping, each configuration receives a fuzzy system with rules explicitly tailored to it. Some resulting rules are shown in Table 4.6.

By again constructing corresponding linguistic variables and a suitable OutputMapper, we can create the final rule file for the suitability approach, which can be looked up at `fuzzyRulesSuitability.frule`.

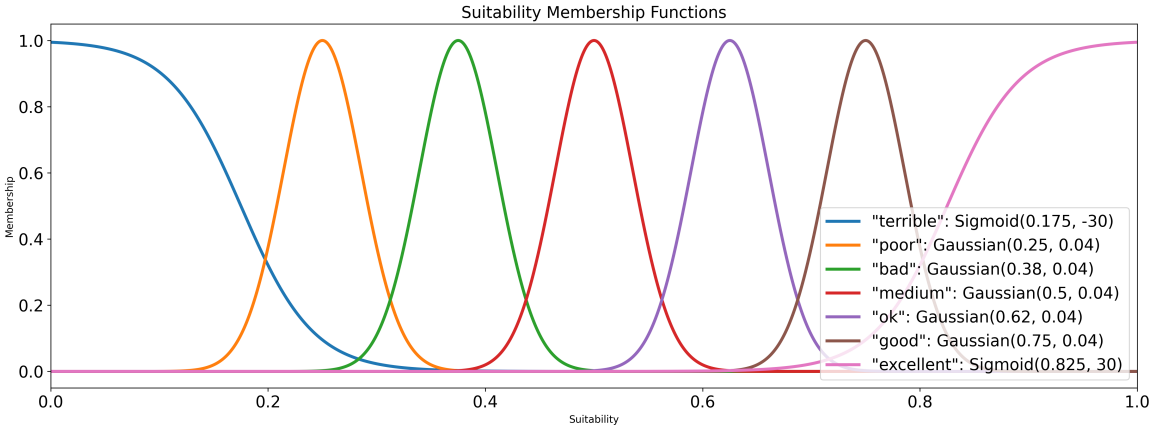


Figure 4.9.: Linguistic variable for the Suitability attribute. The domain between 0% suitability and 100% suitability is divided into 7 classes: `terrible`, `poor`, `bad`, `medium`, `ok`, `good`, and `excellent`. The inner membership functions have a `gaussian` shape, while the outer ones have a `sigmoid` shape to capture the one-sided nature of those boundaries. The choice to use seven classes was made somewhat arbitrarily, with the intention to densely cover the range of suitability values with enough precision.

ParticlesPerCell			Miscellaneous			Configuration				
avg	max	stddev	homo-genity	max-density	threads	Container DataLayout	Traversal	Newton3	Relative speed	Suitability
0.905	15	0.012	0.035	0.531	1	LinkedCells_AoS	lc sliced	enabled	0.450	”bad”
0.944	25	0.012	0.083	0.691	28	VerletClusterLists_AoS	vcl_c06	disabled	0.319	”poor”
0.944	20	0.012	0.079	0.041	12	LinkedCell_SoA	vlc_ sliced	enabled	0.989	”excellent”
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 4.5.: Training data for the Suitability Approach. The dataset contains the LiveInfoData of the simulation, the current configuration, the relative speed, and the suitability values of the configuration. Each row represents a different configuration evaluated in a tuning phase.

Antecedent				Consequent
avgParticlesPC	**homogeneity**	**particlesPCStdDev**	**threadCount**	**Suitability LinkedCells_AoS lc_c01_disabled**
	lower than 0.084	higher than 0.029	higher than 26.0	”medium”
	higher than 0.084	higher than 0.029	higher than 26.0	”bad”
		higher than 0.02	lower than 2.5	”poor”
⋮	⋮	⋮	⋮	⋮

Antecedent				Consequent
maxParticlesPerCell	**homogeneity**	**particlesPCStdDev**	**threadCount**	**Suitability LinkedCells_AoS lc_c04_disabled**
higher than 18.5	lower than 0.082		higher than 18.0 ∧ lower than 26.0	”medium”
higher than 18.5	higher than 0.082		higher than 18.0 ∧ lower than 26.0	”bad”
⋮	⋮	⋮	⋮	⋮

⋮

Table 4.6.: Some extracted fuzzy rules from the decision trees for the Suitability Approach. The rules are grouped by the configuration they predict. The first row is read as: IF (homogeneity = lower than 0.084) ∧ (particlesPerCellStdDev = higher than 0.029) ∧ (threadCount = higher than 26.0) THEN (Suitability LinkedCells_AoS_lc_c01_disabled = ”medium”)

5. Comparison and Evaluation

In this section, we compare the two developed fuzzy tuning approaches with other tuning techniques present in AutoPas and evaluate their performance.

To measure the performance of the fuzzy tuning strategy, we use the scenarios present in `md_flexible` and compare the results with the other tuning strategies of AutoPas. The benchmarks are run on the CoolMUC-2[1] cluster and are repeated with 1, 12, 24, and 28 threads. We use the `timeSpentCalculatingForces` metric to evaluate the performance of the tuning strategies as it gives a good indication of the overall performance of the simulation.

5.1. Exploding Liquid Benchmark (Included in Training Data)

The exploding liquid benchmark simulates a high-density liquid that expands outwards as the simulation progresses. As the data of this scenario was included in the training data, we expect the fuzzy tuning technique to perform well.

The plot in Figure 5.1a shows the time spent calculating the forces for each tuning strategy throughout the simulation. We only include the benchmark results using one thread for brevity, as all thread counts resulted in very similar behavior.

The plot shows that both fuzzy tuning strategies perform close to optimal and show very stable force calculation times throughout the simulation. All other tuning strategies show a much higher variance caused by testing significantly more and worse configurations during the tuning phases.

To show the differences between the strategies in more detail, we also include a boxplot of the time spent calculating the forces for each tuning strategy based on the current phase in Figure 5.1b. All tuning strategies show similar timings during the simulation phases, as they eventually found a close-to-optimal configuration during the tuning phase. The tuning phases, however, differ drastically: All knowledge-based strategies (Fuzzy Tuning and Rule-Based Tuning) tend to perform better in tuning phases and evaluate fewer bad configurations, as the rule files correctly discourage the evaluation of such configurations.

The last plot in Figure 5.1c shows the total time spent calculating the forces for each tuning strategy, again divided into simulation and tuning time. The fuzzy tuning strategies have the lowest total time, with practically no time spent in the tuning phases. In contrast, all other strategies spend about 50% of their total time in the tuning phases.

Both fuzzy tuning approaches perform similarly and are by far the best-performing strategies, achieving a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{Fuzzy[Components]}}} = \frac{32.5s}{16.6s} \approx 1.96$ and $\frac{t_{\text{FullSearch}}}{t_{\text{Fuzzy[Suitability]}}} = \frac{32.5s}{20.3s} \approx 1.60$, respectively.

[1] CoolMUC-2 is a supercomputer located at the Leibniz Supercomputing Centre in Garching, Germany. It consists of 812 Haswell-based nodes with 14 cores each. As a result of hyperthreading, each node supports up to 28 threads. More information can be found at `https://doku.lrz.de/coolmuc-2-11484376.html`

The low tuning overhead is the most significant contributor to the performance of the fuzzy tuning strategies. It is primarily caused by very short and efficient tuning phases, where only a few good configurations are evaluated. This is in contrast to the other tuning strategies, which evaluate many bad configurations during the tuning phases, causing a significant slowdown.

We observe that the suitability approach performs slightly worse during simulation phases compared to all other strategies, as it failed to find the optimal configuration during the first tuning phases. In later tuning phases, however, the suitability approach always succeeds in finding the optimal configuration. Even with this slight error in the suitability approach, it still performed better than all classical strategies.

5.2. Spinodal Decomposition MPI (Related to Training Data)

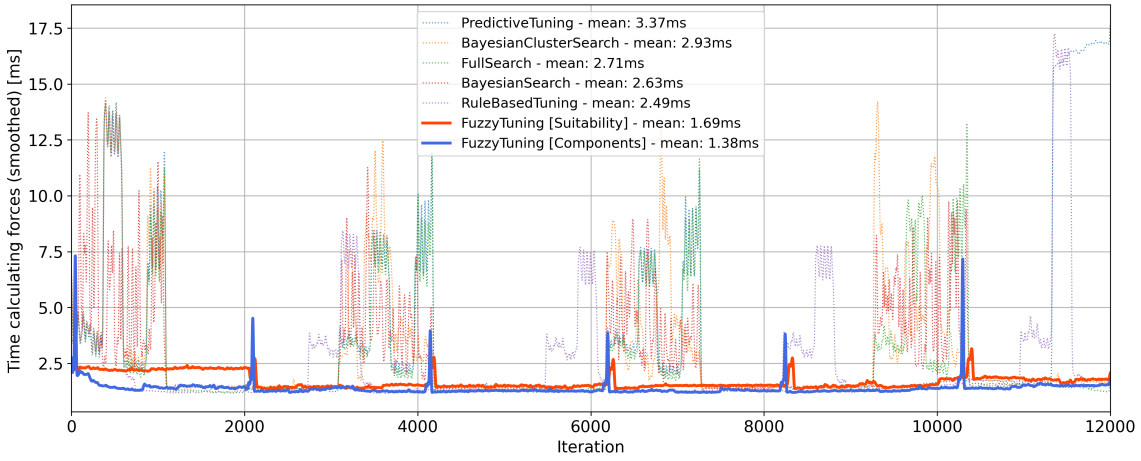
The spinodal decomposition benchmark simulates an unstable liquid that separates into two phases, each having different characteristics. To improve the performance of the simulation, we used four different MPI ranks, each running on 14 threads to simulate the scenario.

As a full Spinodal Decomposition run with just one rank was included in the training data, and the scenario is very homogeneous, we expect the fuzzy tuning strategies to also perform well in the smaller regions handled by the individual MPI ranks, especially as the rule files only contain *relative* rules, that should be unaffected by rescaling the scenario.

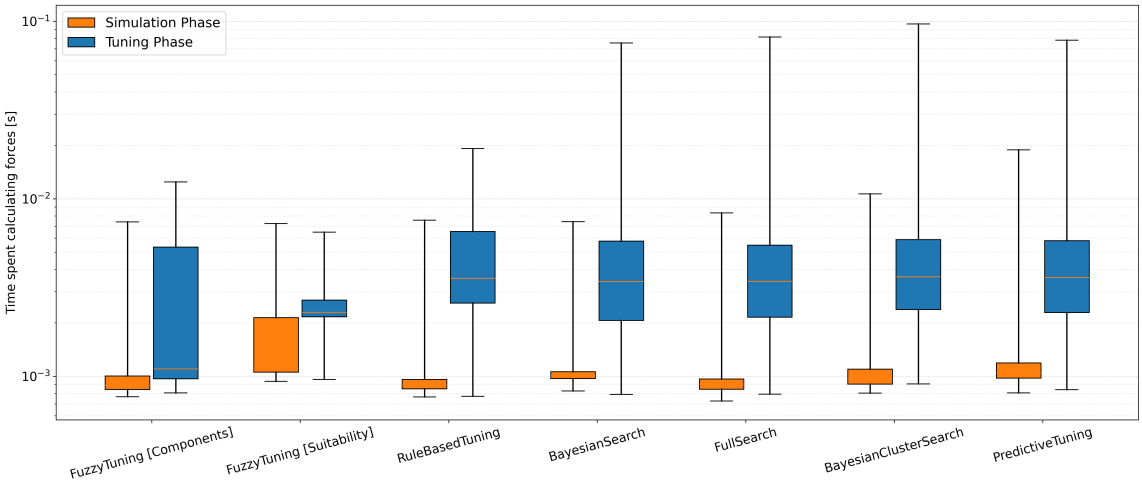
The plot in Figure 5.2a shows the time spent calculating the forces for each tuning strategy throughout the simulation. For brevity, we again only include the benchmark results for the 0th MPI rank, as the results for the other MPI ranks are nearly identical. This time, we see a big difference in both fuzzy tuning strategies, as the component tuning approach performs way better than the suitability approach throughout most of the simulation.

We see that the suitability approach performed the worst during most simulation phases, as it never found the true optimal configuration during the tuning phases. This is primarily caused by the suitability threshold of 10% being too low for this scenario. The low threshold caused the suitability approach to be overly optimistic in its predictions and resulted in very few configurations actually being evaluated during the tuning phases. The predicted configurations were not bad, as shown in Figure 5.2b, but didn't include the optimal configuration. Subsection 5.3.2 will investigate the suitability threshold's effect on the simulation's performance in more detail.

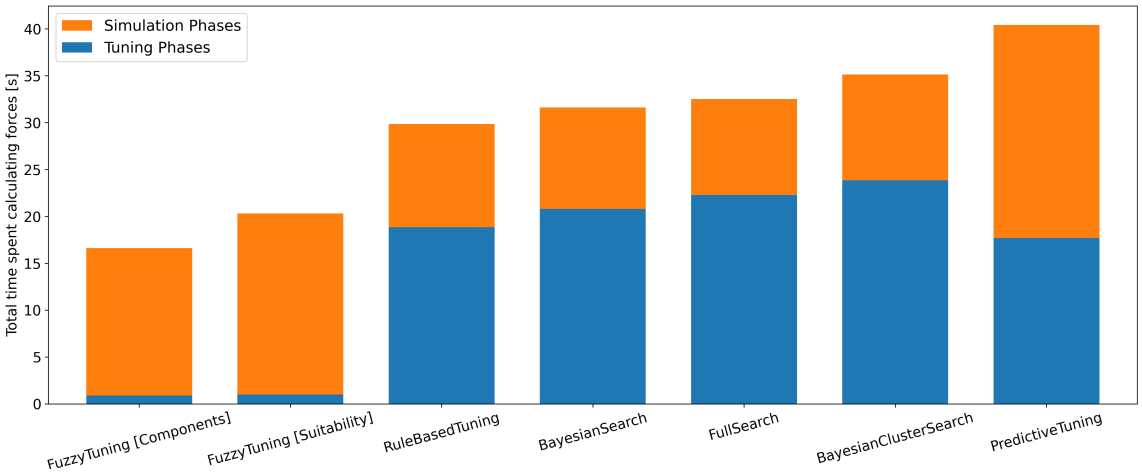
Figure 5.2 shows that the component tuning approach again performs best, with a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{Fuzzy[Components]}}} = \frac{2236.1s}{1650.3s} \approx 1.35$. Surprisingly, the suitability approach performed reasonably well despite never finding the optimal configuration, mainly due to basically no time wasted during the tuning phases. In particular it achieves a speedup of $\frac{t_{\text{FullSearch}}}{t_{\text{Fuzzy[Suitability]}}} = \frac{2236.1s}{1846.1s} \approx 1.21$ which places it on the third place. This shows the importance of efficient tuning phases, as they can cause tremendous overhead if not done correctly.



(a)

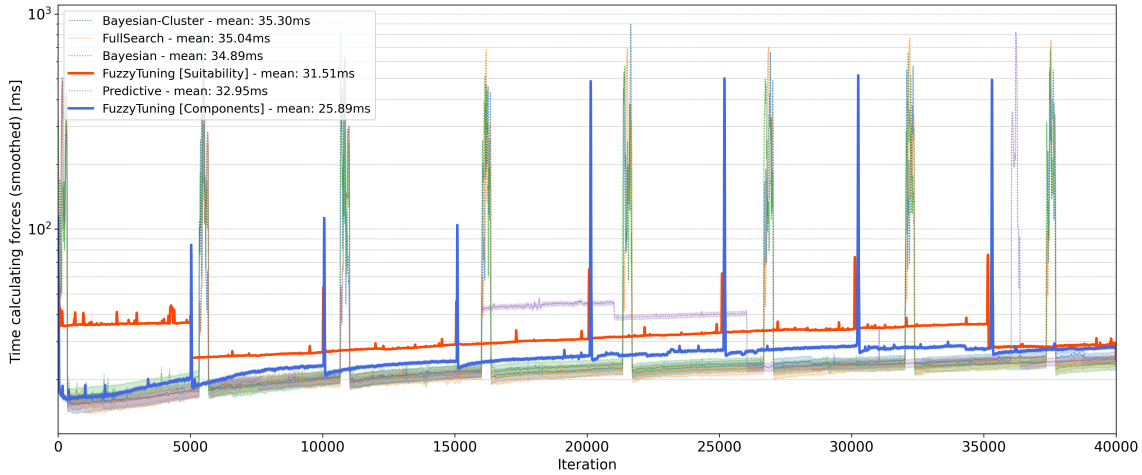


(b)

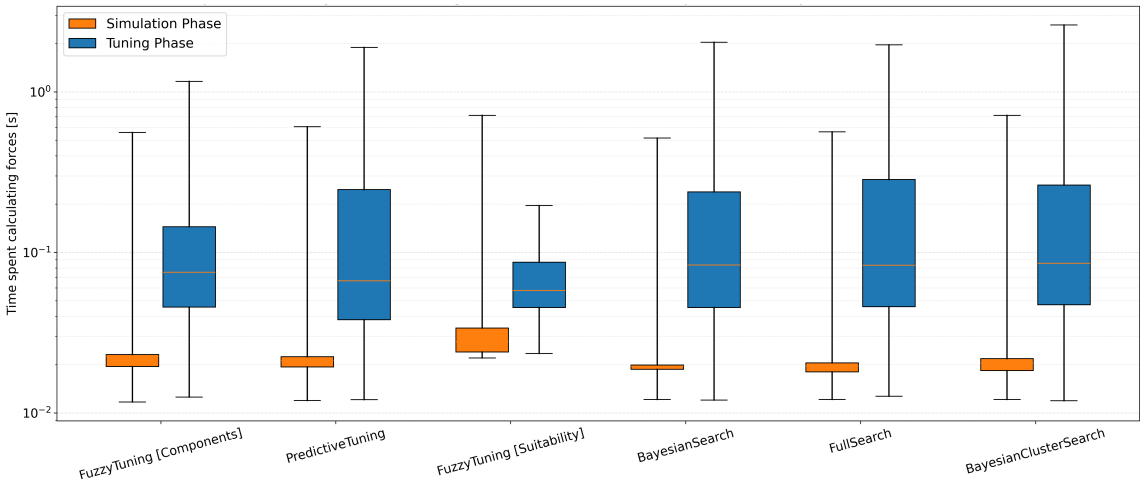


(c)

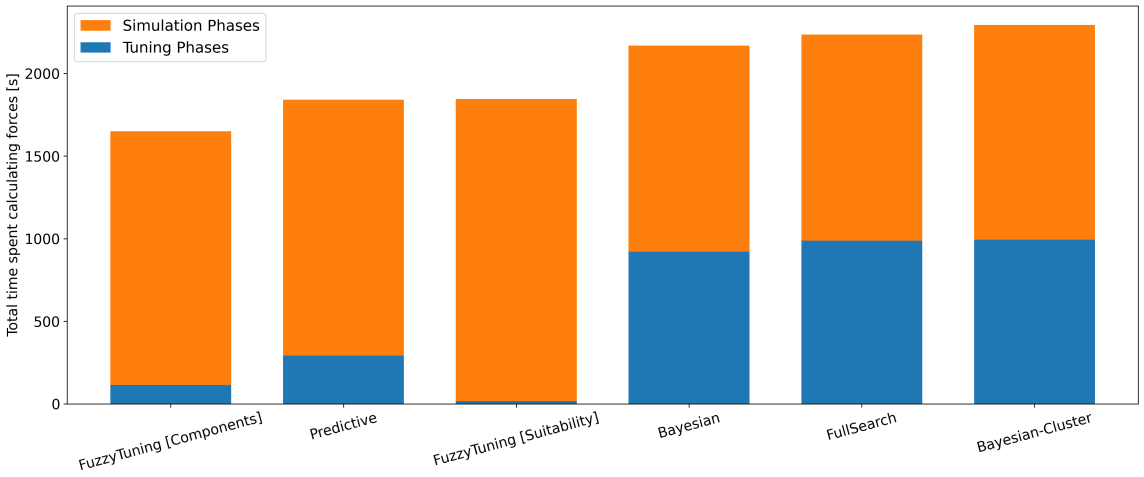
Figure 5.1.: Exploding Liquid benchmark with 1 thread. (a) Time spent calculating forces for every iteration. (b) Boxplots of time spent calculating forces divided into tuning- and simulation phases. (c) Total time spent calculating forces for tuning- and simulation phases. The Suitability approach uses a non-optimal threshold of 10% (see Subsection 5.3.2).



(a)



(b)



(c)

Figure 5.2.: 0th Rank of the Spinodal decomposition benchmark (Total: 4 MPI ranks, 14 threads each). (a) Time spent calculating forces for every iteration. (b) Boxplots of time spent calculating forces divided into tuning- and simulation phases. (c) Total time spent calculating forces for tuning- and simulation phases. The suitability approach uses a non-optimal threshold of 10% (see Subsection 5.3.2).

5.3. Further Analysis

5.3.1. Quality of Predictions During Tuning Phases

As described above, a tremendous slowdown of the classical tuning strategies is caused by very bad configurations encountered during the tuning phases. To further illustrate this, we will investigate the speedup density distribution of all configurations evaluated during the tuning phases of the different strategies. In particular, we will look at the Exploding Liquid and Spinodal Decomposition MPI scenarios introduced above.

The plots in Figure 5.3 show the relative speed distributions of all configurations evaluated during the tuning phases.

All classical tuning strategies tend to encounter configurations with extremely low relative speed during the tuning phases. In the exploding-liquid benchmark, some evaluated configurations are up to 10 times slower than the winning configurations, while we observe iterations up to 100 times slower in the Spinodal Decomposition MPI scenario.

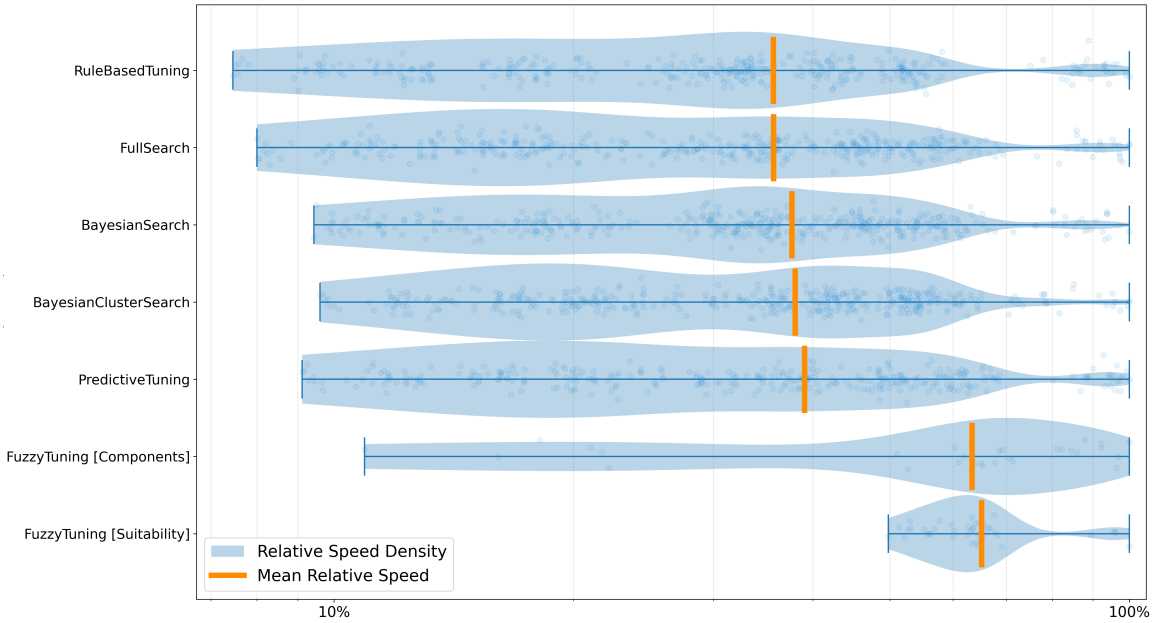
The component tuning approach also encounters such bad configurations, but they are less frequent than in the classical tuning strategies and, therefore, do not significantly impact the overall performance.

The suitability approach performs very well in both scenarios, predicting configurations with the highest median relative speed. However, the high relative speed of the suitability approach could also be caused by the low suitability threshold of 10%, which resulted in the strategy only predicting very few semi-good configurations. As mentioned previously, we will investigate the impact of the suitability threshold in more detail in Subsection 5.3.2.

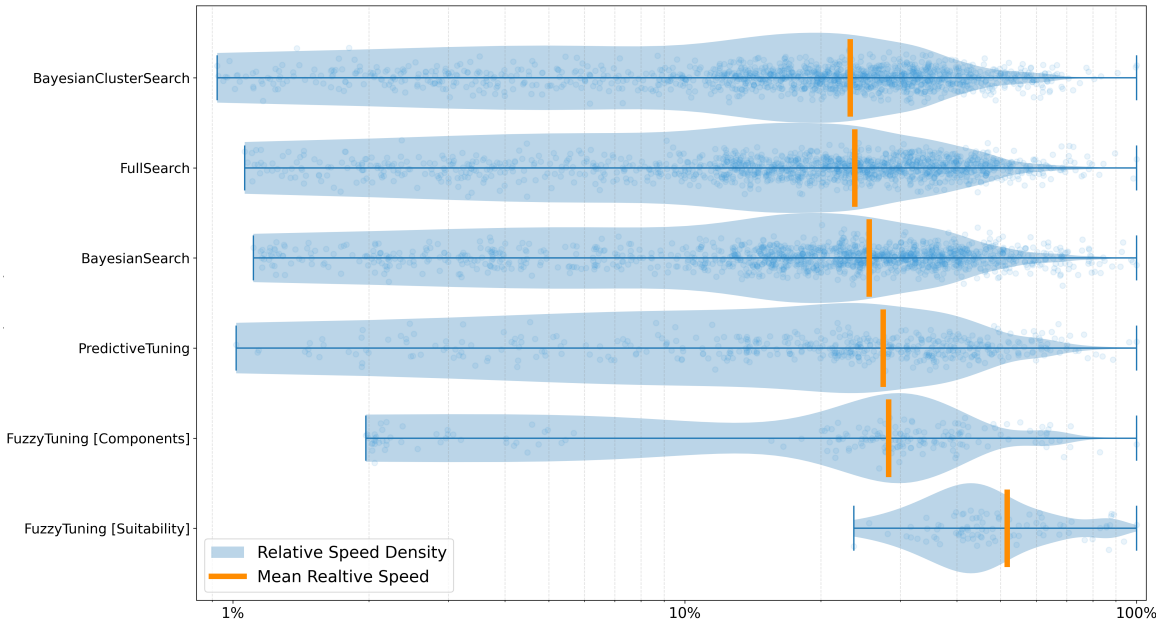
Potential Improvement: Early Stopping of Bad Configurations

As most tuning strategies are plagued by evaluating extremely bad configurations during the tuning phases, it would be highly beneficial to AutoPas if such configurations could be detected during their evaluation and discarded early. Such an improvement could drastically benefit every tuning strategy by significantly reducing the time spent in the tuning phases while still finding the same optimal configuration.

A simple strategy could be aborting a configuration's evaluation if it performs worse than the current best configuration by a particular factor. Additionally, evaluating a configuration can be stopped if it performs worse than the configuration used in the prior simulation phase.



(a) Tuning phases of the Exploding Liquid scenario with one thread.



(b) Tuning phases of rank 0 of the Spinodal Decomposition MPI scenario with 14 threads.

Figure 5.3.: The plot shows the relative speed distribution of all configurations evaluated during the tuning phases of both the Exploding Liquid and Spinodal Decomposition MPI scenarios calculated from the smoothed timings (see Section A.1). The fuzzy tuning strategies generally encounter better configurations during the tuning phases, which improves their total performance, as less time is spent evaluating bad configurations.

5.3.2. Optimal Suitability Threshold

In previous measurements, the Component tuning approach performed better than the Suitability tuning approach, mainly due to the suitability approach not finding the optimal configuration during the tuning phases (see Figure 5.2a).

The previous benchmarks were executed with a rule file specifying that only the top 10% of configurations with the highest suitability should be selected, which may have been too low, causing a high chance of not finding the globally optimal configuration.

To investigate this issue, we reran the Exploding Liquid benchmark with different suitability thresholds each time, measuring the total runtimes as shown in Figure 5.4.

From the figure, we observe that very low thresholds perform poorly, as they select too few configurations to be evaluated in the tuning phases, resulting in a high chance of not finding the optimal configuration. Very high thresholds also perform poorly, as high suitability values cause the strategy to behave like FullSeach, selecting nearly all configurations to be evaluated.

The optimal suitability threshold for this scenario lies between 20% and 40%, which guarantees that the best configuration is selected for the tuning phases while still keeping the total number of evaluated configurations low. Following this observation, we updated the default value of the rule file of the suitability approach to 30%. This change should improve the suitability approach's performance in future benchmarks.

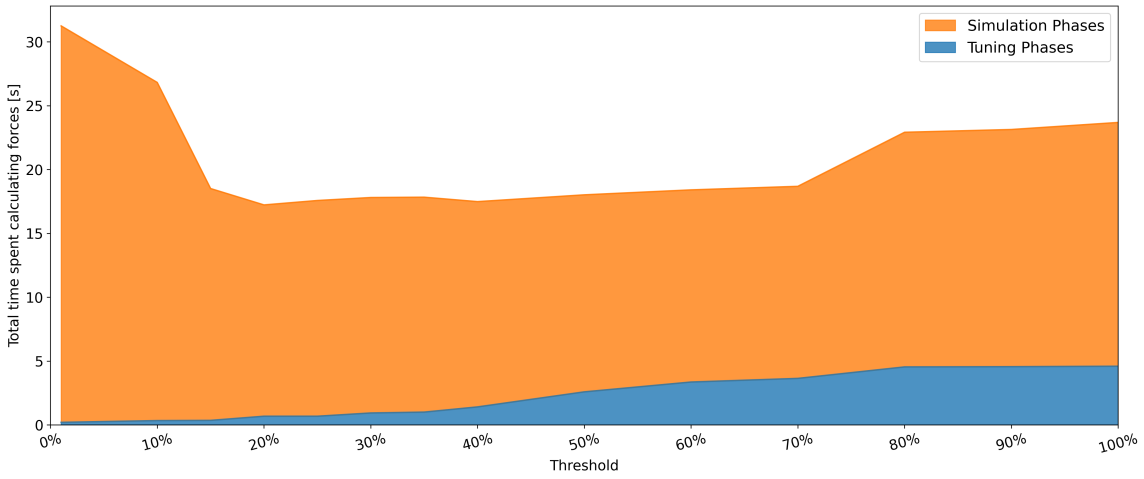


Figure 5.4.: Exploding liquid benchmark with different suitability thresholds. The fastest runtimes are achieved with a threshold between 20% and 40%. The time spent calculating forces during tuning phases increases with higher thresholds as the strategy converges towards the behavior of FullSearch.

5.3.3. Generalization of Rule Extraction Process

Previous measurements of the Exploding Liquid and the Spinodal Decomposition MPI benchmark showed that the fuzzy tuning strategies perform well. However, all previously tested scenarios were in some way included in the training data, which could have biased the results in favor of the fuzzy tuning strategies.

To investigate the generalization of the rule extraction process, we reran the rule extraction process without the Exploding Liquid scenario being present in the training data.

To test the resulting impact, we reran the Exploding Liquid benchmark with the new *holdout*-rule files to see if the fuzzy tuning strategies can still perform well even without the scenario being directly included in the training data.

The results in Figure 5.5 show that the fuzzy tuning strategies still perform remarkably well, with comparable performance to the previous measurements.

The remaining training scenarios provided enough general tuning information to still allow the fuzzy tuning strategies to extrapolate the optimal configuration for the Exploding Liquid scenario.

Therefore, we conclude that the rule extraction process is reasonably robust, and the extracted rules can be generalized to similar scenarios, even if they were not included in the training data.

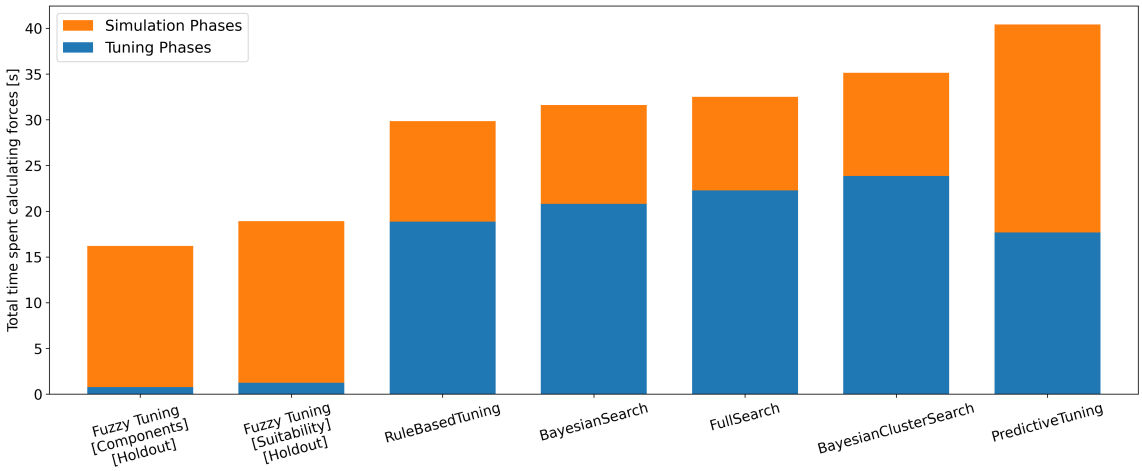


Figure 5.5.: Total time spent calculating forces for tuning- and simulation phases for the Exploding Liquid benchmark without the scenario being included in the training data. The fuzzy tuning strategies still perform best, and the timings are comparable to those of the previous measurements.

6. Future Work

In this chapter, we discuss some of the possible improvements that could be made to the current system to improve its performance and usability.

6.1. Dynamic Rule Generation

The current rule extraction process is static and requires a pre-collected dataset of selected scenarios. This is an obvious limitation, as users cannot be expected to collect a dataset of scenarios prior to using the library. Another issue is that the generated rules can only be expected to perform well in scenarios similar to the ones in the dataset, preventing the rules from being shared between vastly different use cases.

One could look into ways of adaptively updating the expert knowledge as new scenarios are encountered. This could be done by spending extra time during the simulation to evaluate the performance data of recently or additionally executed configurations and updating the expert knowledge on the fly.

6.2. Improving Tuning Strategies

As previously discussed in subsubsection 5.3.1, all current tuning strategies suffer from evaluating extremely bad configurations during the tuning phases. Especially for strategies without knowledge bases, this is a significant issue and causes enormous tuning overhead that could be avoided. Implementing an early stopping mechanism could drastically improve all tuning strategies and thus benefit the core idea of AutoPas.

6.3. Simplification of the Fuzzy System to Decision Trees

As the current way of generating the Fuzzy Systems with the data-driven approach already makes use of Decision Trees, one could look into ways of directly using Decision Trees to perform the tuning.

The current implementation of the Component Tuning approach is already quite close to a Decision Tree approach as it uses the MOM defuzzification. Since this approach showed promising results, it is possible that the Fuzzy Tuning approach could be simplified to a Decision Tree approach without losing much performance. This would make the tuning process more transparent and easier for users to understand, as the complexity introduced by fuzzy sets and membership functions could be avoided.

To test this hypothesis using the current system, one could change membership functions to crisp splits as originally depicted in Figure 4.2 and rerun the benchmarks to see if the performance is still comparable. Such a rule file would emulate (Crisp) Decision Trees, following the same rules but without the fuzziness.

7. Conclusion

This thesis introduced a novel fuzzy logic-based tuning strategy for AutoPas and implemented a generic Fuzzy Tuning Library into the AutoPas framework, providing a reusable foundation for future research projects.

A key contribution was the development of a data-driven approach to automatically generate competitive fuzzy systems, enabling the tuning of complex systems without extensive prior knowledge. The results demonstrated that the proposed Fuzzy Tuning Strategy significantly outperformed existing tuning strategies on selected benchmarks, substantially reducing the total simulation runtime by up to a factor of 1.96.

While the fuzzy tuning strategy and the data-driven rule generation process showed promise, they are not universal solutions, as considerable upfront effort is necessary to collect the data to generate the rule base. Since such a workload cannot be expected from typical users of AutoPas, future research is needed to streamline the data collection and rule generation process to make the fuzzy tuning strategy more accessible to a broader audience.

In conclusion, the Fuzzy Tuning Strategy and the data-driven rule generation process represent a significant step forward in tuning AutoPas simulations and offer a solid foundation for future research. While challenges remain in making it more accessible and broadly applicable, the potential for substantial performance gains makes this an exciting area for continued investigation and refinement.

A. Appendix

A.1. TuningDataLogger Fields

The following fields are currently available in the TuningData file. The TuningData file is a CSV file containing the performance information and configuration parameters during the tuning phases. This data is used to guide the data-driven rule generation process for the Fuzzy Tuning Strategy. The data is collected and logged by the `TuningDataLogger` class of the AutoPas library.

Date	The date and time when the data was collected.
Iteration	The current iteration number of the simulation.
Container	The type of container used to store the particles in the simulation (e.g., LinkedCells, VerletLists).
CellSizeFactor	A factor that determines the size of the cells relative to the cutoff radius.
Traversal	The method used to traverse the cells and calculate interactions between particles.
Load Estimator	The strategy used to estimate and balance the computational load across different parts of the simulation domain.
Data Layout	The arrangement of particle data in memory (e.g., AoS for Array of Structures, SoA for Structure of Arrays).
Newton 3	Indicates whether Newton's third law optimization is used to reduce computational effort (enabled/disabled).
Reduced	The reduced performance data for configuration is calculated by aggregating its timing data across all its evaluated iterations. The specific aggregation method can be configured via the `.yaml` configuration file.
Smoothed	A smoothed version of the reduced performance data.

A.2. LiveInfoLogger Fields

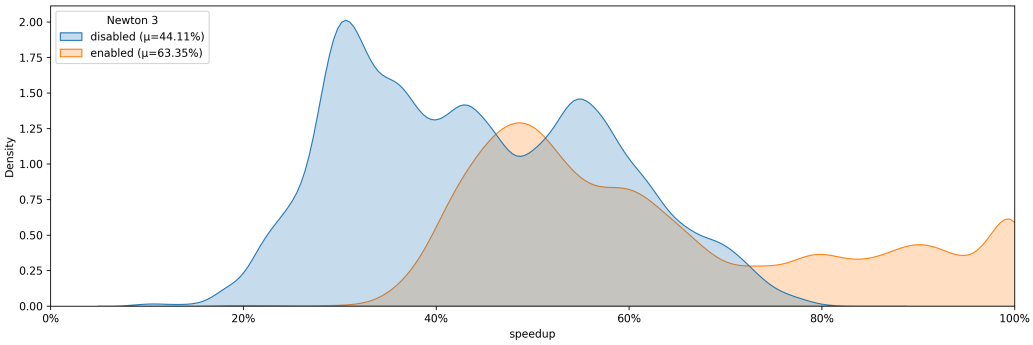
The following fields are currently available in the LiveInfoData file. The LiveInfoData file is a CSV file containing summary statistics about the simulation state at each iteration. In the current implementation, this data is the only source of information for the Fuzzy Tuning

Strategy to make decisions during the simulation. The data is collected and logged by the `LiveInfoLogger` class of the AutoPas library.

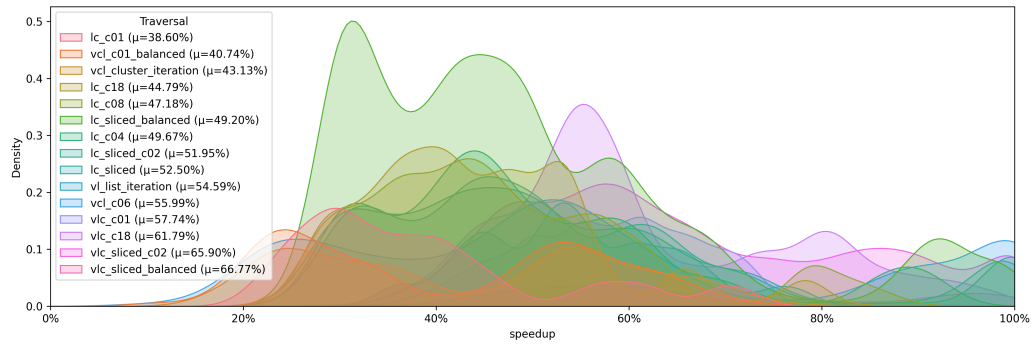
Iteration	The current iteration number of the simulation.
avgParticlesPerCell	The average number of particles per cell in the simulation domain.
cutoff	The cutoff radius for the interaction of particles, beyond which particles do not interact.
domainSizeX	The size of the simulation domain in the X dimension.
domainSizeY	The size of the simulation domain in the Y dimension.
domainSizeZ	The size of the simulation domain in the Z dimension.
estimatedNumNeighborInteractions	The estimated number of neighbor interactions between all particles in the simulation domain.
homogeneity	A measure of the distribution uniformity of particles across the cells.
maxDensity	The maximum density of particles in any cell.
maxParticlesPerCell	The maximum number of particles found in any single cell.
minParticlesPerCell	The minimum number of particles found in any single cell.
numCells	The total number of cells in the simulation domain.
numEmptyCells	The number of cells that contain no particles.
numHaloParticles	The number of particles in the halo region (boundary region) of the simulation domain.
numParticles	The total number of particles in the simulation domain.
particleSize	The number of bytes used to store a single particle in memory.
particleSizeNeededByFunctor	The particle size required by the functor (the function used for calculating interactions).
particlesPerBlurredCellStdDev	The standard deviation of the number of particles per blurred cell provides a measure of particle distribution variability.
particlesPerCellStdDev	The standard deviation of the number of particles per cell, indicating the variability in particle distribution.
rebuildFrequency	The frequency at which the neighbor list is rebuilt.
skin	The skin width is added to the cutoff radius to create a buffer zone for neighbor lists, ensuring efficient interaction calculations.
threadCount	The number of threads used for parallel processing in the simulation.

A.3. Density Plots of Relative Speed present in the Dataset

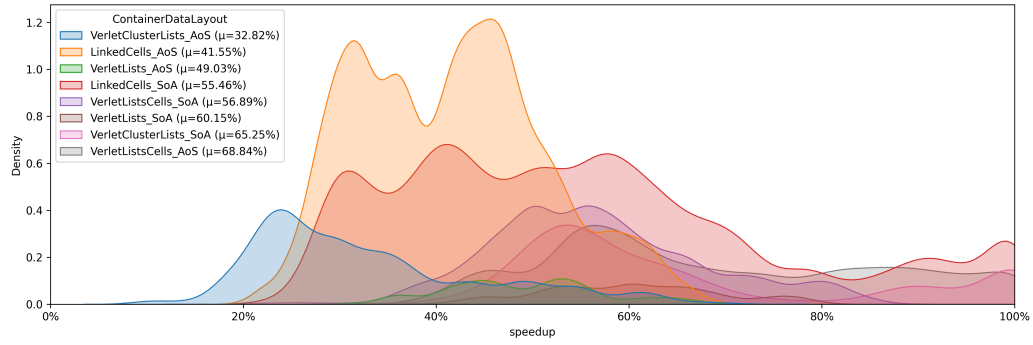
To investigate the collected data, we performed some exploratory data analysis to identify patterns and trends in the dataset. We created density plots to visualize the distribution of the relative speed based on different configuration options.



(a) Distribution of the relative speed based on the Newton 3 option. We can see that enabling Newton3 is generally the better option, allowing for higher relative speeds. Therefore, we can confirm that Newton 3 is generally a good option to enable.



(b) Distribution of the relative speed based on the Traversal option. The vlc_sliced_balanced option generally performed better than the other options with an expected relative speed of 66%.



(c) Distribution of the relative speed based on the ContainerDatalayout option. The VerletListCells_AoS ContainerDatalayout performed best with an expected relative speed of 68.8%.

Figure A.1.: Density plots showing the relative speed distribution based on different configuration options for the collected dataset.

List of Figures

List of Tables

Bibliography

[BMK96] Bernadette Bouchon-Meunier and Vladik Kreinovich. Axiomatic description of implication leads to a classical formula with logical modifiers: (in particular, mamdani's choice of "and" as implication is not so weird after all). 1996.

[BPR+16] Tobias Brink, Martin Peterlechner, Harald Rösner, Karsten Albe, and Gerhard Wilde. Influence of crystalline nanoprecipitates on shear-band propagation in cu-zr-based metallic glasses. *Phys. Rev. Appl.*, 5:054005, May 2016.

[CBMO06] Keeley Crockett, Zuhair Bandar, David Mclean, and James O'Shea. On constructing a fuzzy inference framework using crisp decision trees. *Fuzzy Sets and Systems*, 157(21):2809–2832, 2006.

[Che] Chemie.de. Lennard-jones-potential. `https://www.chemie.de/lexikon/Lennard-Jones-Potential.html`. Accessed: 2024-07-07.

[GSBN21] 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, 2021.

[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.

[Iri21] Karl Irikura. Physics-guided curve fitting for potential-energy functions of diatomic molecules. *Authorea*, April 2021.

[JPRS06] E. Jodoin, C.A. Pena Reyes, and E. Sanchez. A method for the fuzzification of categorical variables. In *2006 IEEE International Conference on Fuzzy Systems*, pages 831–838, 2006.

[LM15] Benedict Leimkuhler and Charles Matthews. *Molecular Dynamics: With Deterministic and Stochastic Numerical Methods*. Interdisciplinary Applied Mathematics. Springer, May 2015.

[MAMM20] C Y Maghfiroh, A Arkundato, Misto, and W Maulina. Parameters (sigma, epsilon) of lennard-jones for fe, ni, pb for potential and cr based on melting point values using the molecular dynamics method of the lammps program. *Journal of Physics: Conference Series*, 1491(1):012022, October 2020.

[MKEC22] Ali Mohammed, Jonas H. Müller Korndörfer, Ahmed Eleliemy, and Florina M. Ciorba. Automated scheduling algorithm selection and chunk parameter calculation in openmp. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):4383–4394, 2022.

[Mur12] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[NGNB23] Samuel James Newcome, Fabio Alexander Gratl, Philipp Neumann, and Hans-Joachim Bungartz. Towards auto-tuning multi-site molecular dynamics simulations with autopas. *Journal of Computational and Applied Mathematics*, 433:115278, 2023.

[Phy] Nexus Physics. The lennard-jones potential. `https://www.compadre.org/nexusph/course/The_Lennard-Jones_Potential`. Accessed: 2024-07-07.

[PS17] Juan R. Perilla and Klaus Schulten. Physical properties of the hiv-1 capsid from all-atom molecular dynamics simulations. *Nature Communications*, 8(1):15959, 2017.

[RdCC12] Pilar Rey-del Castillo and Jesús Cardeñosa. Fuzzy min-max neural networks for categorical data: application to missing data imputation. *Neural Computing and Applications*, 21(7):1349–1362, 2012.

[VBC08] G. Viccione, V. Bovolin, and E. Pugliese Carratelli. Defining and optimizing algorithms for neighbouring particle identification in sph fluid simulations. *International Journal for Numerical Methods in Fluids*, 58(6):625–638, 2008.

[ZBB+13] Franck Zielinski, Pierre Baudin, Gérard Baudin, Pierre Baudin, and Gérard Baudin. Quantum states of atoms and molecules. *Chemical Education Digital Library*, 1(1):1–10, 2013.